



Pruebas de Software

ÍNDICE

	Página
Presentación	4
Red de contenidos	5
Unidad de aprendizaje 1: Fundamentos de Pruebas de Software	
1.1 Tema 1 : Introducción a TDD	7
1.1.1. : TDD y Tipos de Pruebas	7
1.1.2. : Pruebas Unitarias	12
1.2 Tema 2 : Pruebas Unitarias con el framework JUnit	15
1.2.1. : Creación y Ejecución de una prueba unitaria simple con JUnit en Netbeans o Eclipse	16
1.2.2. : Creación y Ejecución de una prueba unitaria de regular complejidad con JUnit en Netbeans o Eclipse	19
1.2.3. : Creación y Ejecución de una prueba unitaria avanzada con JUnit en Netbeans o Eclipse	24
Unidad de aprendizaje 2: Fundamentos Rational Functional Tester	
2.1 Tema 3 : Introducción al Rational Functional Tester	32
2.1.1. : Arquitectura de Rational Functional Tester	32
2.1.2. : Configuración del entorno de pruebas	34
2.1.3. : Configuración de aplicaciones Java a probar	40
2.1.4. : Proyectos de pruebas funcionales en Rational Functional Tester	45
2.2 Tema 4 : Script de pruebas funcionales	48
2.2.1. : Grabación de un script	49
2.2.2. : Reproducción de un script	67
2.2.3. : Revisión de los resultados	68
2.2.4. : Características avanzadas de script de pruebas	68
Unidad de aprendizaje 3: Fundamentos Rational Performance Tester	
3.1 Tema 5 : Introducción al Rational Performance Tester	76
3.1.1. : Arquitectura de Rational Performance Tester	76

3.1.2. : Características y beneficios	77
3.2 Tema 6 : Pruebas de rendimiento	83
3.2.1. : Crear y ejecutar pruebas de rendimiento	
3.2.2. : Análisis de resultados	
4.1.3. : Modificar pruebas de rendimiento	

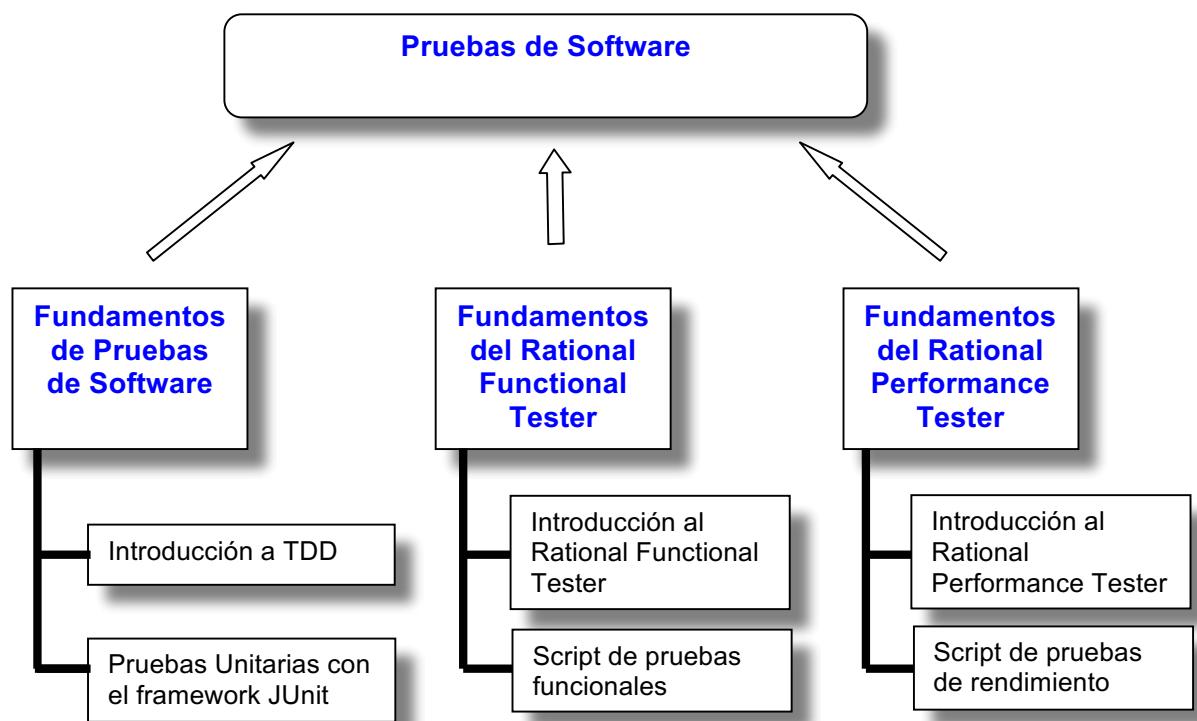
PRESENTACIÓN

Pruebas de Software pertenece a la línea de carrera y se dicta en la carrera profesional de Computación e Informática. Brinda los conceptos básicos relacionados al área de control de calidad de software y administración de pruebas de software, alineados a las mejores prácticas en desarrollo de software.

El manual para el curso ha sido diseñado bajo la modalidad de unidades de aprendizaje, las que se desarrollan durante semanas determinadas. En cada una de ellas, hallará los logros, que debe alcanzar al final de la unidad; el tema tratado, el cual será ampliamente desarrollado; y los contenidos, que debe desarrollar, es decir, los subtemas. Por último, encontrará las actividades que deberá desarrollar en cada sesión, que le permitirán reforzar lo aprendido en la clase.

El curso es eminentemente práctico: consiste en sesiones teóricas acompañadas con aplicaciones prácticas. En primer lugar, se explica la importancia de la verificación y validación de software para el control de calidad del producto de software y se presenta una introducción al Test Driven Development. Continúa con la presentación de los fundamentos del Rational Functional Tester para la creación de scripts de pruebas funcionales. Por último, se concluye con la aplicación del Rational Performance Tester para el diseño, creación, ejecución y análisis de pruebas de rendimiento.

RED DE CONTENIDOS



UNIDAD DE
APRENDIZAJE

1

FUNDAMENTOS DE PRUEBAS DE SOFTWARE

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, el alumno reconoce la importancia de la validación y verificación de software para el control de calidad del producto de software. Asimismo asimila los conceptos y aplica Test Driven Development (TDD) en una aplicación Java

TEMARIO

1.1. Tema 1: Introducción a TDD

- 1.1.1. TDD y Tipos de Pruebas
 - 1.1.1.1. TDD y su utilidad
 - 1.1.1.2. Tipos de Pruebas
 - 1.1.1.2.1. Pruebas de Aceptación
 - 1.1.1.2.2. Pruebas Funcionales
 - 1.1.1.2.3. Pruebas de Sistema
 - 1.1.1.2.4. Pruebas Unitarias
 - 1.1.1.2.5. Pruebas de Integración
- 1.1.2. Pruebas Unitarias
 - 1.1.2.1. Las pruebas unitarias y su utilidad

1.2. Tema 2: Pruebas Unitarias con el framework JUnit

- 1.2.1. Creación y Ejecución de una prueba unitaria simple con JUnit en Netbeans o Eclipse
- 1.2.2. Creación y Ejecución de una prueba unitaria de regular complejidad con JUnit en Netbeans o Eclipse
- 1.2.3. Creación y Ejecución de una prueba unitaria avanzada con JUnit en Netbeans o Eclipse

ACTIVIDADES PROPUESTAS

- Los alumnos comprenden el significado y la utilidad del TDD asimismo crean y ejecutan pruebas unitarias de un caso de uso a partir de su especificación, un prototipo o validaciones en el ingreso de datos.

1.1. INTRODUCCIÓN A TDD

El Desarrollo Dirigido por Tests (Test Driven Development), al cual nos referiremos como TDD, es una técnica de diseño e implementación de software incluida dentro de la metodología XP (eXtreme Programming). TDD es una técnica para diseñar software que se centra en tres pilares fundamentales:

- La implementación de las funciones justas que el cliente necesita y no más.
- La minimización del número de defectos que llegan al software en fase de producción.
- La producción de software modular, altamente reutilizable y preparado para el cambio.

1.1.1. TDD y Tipos de pruebas

1.1.1.1. TDD y su utilidad.

TDD es la respuesta a las grandes preguntas de:

¿Cómo lo hago?, ¿Por dónde empiezo?, ¿Cómo sé qué es lo que hay que implementar y lo que no?, ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente?

Hasta ahora estábamos acostumbrados a que los casos de uso, eran las unidades de trabajo más pequeñas sobre las que ponerse a desarrollar código. Con TDD intentamos traducir el caso de uso en X ejemplos, hasta que el número de ejemplos sea suficiente como para describir la tarea sin lugar a malinterpretaciones de ningún tipo.

Kent Beck (considerado uno de los padres de la metodología XP) enumera los siguientes beneficios de aplicar TDD:

- La calidad del software aumenta.
- Conseguimos código altamente reutilizable.
- El trabajo en equipo se hace más fácil, une a las personas.
- Nos permite confiar en nuestros compañeros aunque tengan menos experiencia.
- Multiplica la comunicación entre los miembros del equipo.
- Las personas encargadas de la garantía de calidad adquieren un rol más inteligente e interesante.
- Escribir el test antes que el código nos obliga a escribir el mínimo de funcionalidad necesaria, evitando sobrediseñar.
- Cuando revisamos un proyecto desarrollado mediante TDD, nos damos cuenta de que los tests son la mejor documentación técnica que podemos consultar a la hora de entender qué misión cumple cada pieza del rompecabezas.

La esencia de TDD es sencilla pero ponerla en práctica correctamente es cuestión de entrenamiento. El algoritmo TDD sólo tiene tres pasos:

- Escribir la especificación del requisito (el test).
- Implementar el código según dicho ejemplo.
- Refactorizar para eliminar duplicidad y hacer mejoras.

TDD / Pasos



¿Qué queremos que haga el código que vamos a desarrollar?

¿Qué hechos demostrarán que nuestro código funciona?

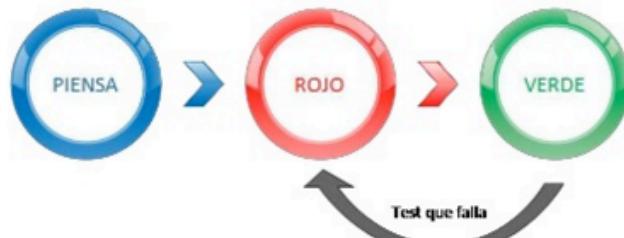
TDD / Pasos



¿Escribimos el test?

Pensamos en el comportamiento del código y su interface público

TDD / Pasos



Escribimos el código de producción

SOLO el código necesario para pasar el test

TDD / Pasos



Refactorizamos la funcionalidad desarrollada

Durante la refactorización NO DEBEMOS cambiar la semántica (sentido)

Fuente: <http://es.slideshare.net/FernandoPrez16/introducción-a-tdd-y-unit-testing>

Figura 1.1. TDD/ Pasos

1.1.1.2. Tipos de Pruebas

La nomenclatura sobre tests puede ser francamente caótica. Ha sido fuente de discusión en los últimos años y sigue sin existir universalmente de manera categórica. Cada equipo tiende a adoptar sus propias convenciones, ya que existen distintos aspectos a considerar para denominar tests. Por aspecto quiere decir que, según cómo se mire, el test se puede clasificar de una manera o de otra. Así se habla, por ejemplo, del aspecto visibilidad (si se sabe lo que hay dentro del SUT - System Under Test - o no), del aspecto potestad (a quién pertenece el test), etc.

Desde el aspecto potestad, es decir, mirando los tests según a quién le pertenecen, distinguimos entre tests escritos por desarrolladores y tests escritos por el Dueño del Producto. Recordemos que el Dueño del Producto es el analista de negocio o bien el propio cliente. Lo ideal es que el analista de negocio ayude al cliente a escribir los tests para asegurarse de que las afirmaciones están totalmente libres de ambigüedad. Los tests que pertenecen al Dueño del Producto se llaman tests de cliente o de aceptación.

En el siguiente diagrama se muestra la clasificación de los tests típica de un entorno TDD. A la izquierda, se agrupan los tests que pertenecen a desarrolladores y, a la derecha, los que pertenecen al Dueño del Producto. A su vez, algunos tipos de tests contienen a otros.

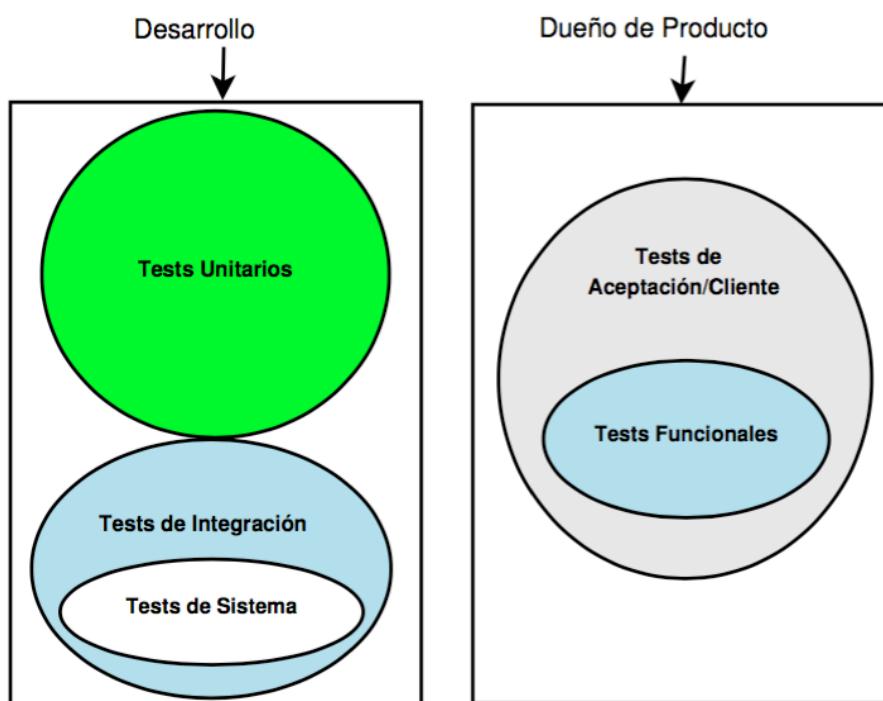


Figura 1.2. Tipos de Pruebas

1.1.1.2.1. Pruebas de Aceptación.

Es un test que permite comprobar que el software cumple con un requisito de negocio. Un test de aceptación es un ejemplo escrito

con el lenguaje del cliente pero que puede ser ejecutado por la máquina. Algunos ejemplos:

- El producto X con precio S/ 50 tiene un precio final de S/ 59 después de aplicar el impuesto Z
- Si el paciente nació el 1 de junio de 1981, su edad es de 28 años en agosto de 2009

¿Los tests de aceptación no usan la interfaz de usuario del programa? Podría ser que sí, pero en la mayoría de los casos la respuesta debe ser no. Los tests de carga y de rendimiento son de aceptación cuando el cliente los considera requisitos de negocio. Si el cliente no los requiere, serán tests de desarrollo.

1.1.1.2.2. Pruebas Funcionales.

Todos los tests son en realidad funcionales, puesto que todos ejercitan alguna función del SUT, aunque en el nivel más elemental sea un método de una clase. No obstante, cuando se habla del aspecto funcional, se distingue entre test funcional y test no funcional. Un test funcional es un subconjunto de los tests de aceptación. Es decir, comprueban alguna funcionalidad con valor de negocio. Hasta ahora, todos los tests de aceptación que hemos visto son tests funcionales. Los tests de aceptación tienen un ámbito mayor porque hay requerimientos de negocio que hablan de tiempos de respuesta, capacidad de carga de la aplicación, etc; cuestiones que van más allá de la funcionalidad. Un test funcional es un test de aceptación pero, uno de aceptación, no tiene por qué ser funcional.

1.1.1.2.3. Pruebas de Sistema.

Es el mayor de los tests de integración, ya que integra varias partes del sistema. Se trata de un test que puede ir, incluso, de extremo a extremo de la aplicación o del sistema. Se habla de sistema porque es un término más general que aplicación, pero no se refiere a administración de sistemas, no es que estemos probando el servidor web o el servidor SMTP aunque, tales servicios, podrían ser una parte de nuestro sistema. Así pues, un test del sistema se ejerce tal cual lo haría el usuario humano, usando los mismos puntos de entrada (aquí sí es la interfaz gráfica) y llegando a modificar la base de datos o lo que haya en el otro extremo. ¿Cómo se puede automatizar el uso de la interfaz de usuario y validar que funciona? Hay software que permite hacerlo. Por ejemplo, si la interfaz de usuario es web, el plugin Selenium para el navegador Mozilla Firefox nos permite registrar nuestra actividad en una página web como si estuviéramos grabando un vídeo para luego reproducir la secuencia automáticamente y detectar cambios en la respuesta del sitio web. Pongamos que grabo la forma en que relleno un formulario con una dirección de correo electrónico incorrecta para que el sitio web me envíe un mensaje de error. Cada vez que quiera volver a comprobar que el sitio web responde igual ante esa entrada, sólo tengo que ejecutar el test generado por

Selenium. Hay herramientas que permiten hacer lo mismo mediante programación: nos dan una API para seleccionar controles gráficos, y accionarlos desde código fuente, comprobando el estado de la ejecución con sentencias condicionales o asertivas. El propio Selenium lo permite.

Sirva el ejemplo para recalcar que no hay una sola herramienta ni forma de escribir tests de sistema, más bien depende de cada sistema.

Los tests de sistema son muy frágiles en el sentido de que cualquier cambio en cualquiera de las partes que componen el sistema, puede romperlos. No es recomendable escribir un gran número de ellos por su fragilidad. Si la cobertura de otros tipos de tests de granularidad más fina, como por ejemplo los unitarios, es amplia, la probabilidad de que los errores sólo se detecten con tests de sistema es muy baja. O sea, que si hemos ido haciendo TDD, no es productivo escribir tests de sistema para todas las posibles formas de uso del sistema, ya que esta redundancia se traduce en un aumento del costo de mantenimiento de los tests. Por el contrario, si no tenemos escrito absolutamente ningún tipo de test, blindar la aplicación con tests de sistema será el paso más recomendable antes de hacer modificaciones en el código fuente. Luego, cuando ya hubiesen tests unitarios para los nuevos cambios introducidos, se podrían ir desechando tests de sistema.

1.1.1.2.4. Pruebas Unitarias.

Son los tests más importantes para el practicante TDD, los ineludibles. Cada test unitario o test unidad (unit test en inglés) es un paso que andamos en el camino de la implementación del software. Todo test unitario debe ser:

- Atómico
- Independiente
- Inocuo
- Rápido

Si no cumple estas premisas entonces no es un test unitario, aunque se ejecute con una herramienta tipo xUnit.

Atómico significa que el test prueba la mínima cantidad de funcionalidad posible. Esto es, probará un solo comportamiento de un método de una clase. El mismo método puede presentar distintas respuestas ante distintas entradas o distinto contexto. El test unitario se ocupará exclusivamente de uno de esos comportamientos, es decir, de un único camino de ejecución. A veces, la llamada al método provoca que internamente se invoque a otros métodos; cuando esto ocurre, decimos que el test tiene menor granularidad, o que es menos fino. Lo ideal es que los tests unitarios ataquen a métodos lo más planos posibles, es decir, que prueben lo que es indivisible. La razón es que un test atómico nos evita tener que usar el depurador para encontrar un defecto en el SUT, puesto que su causa será muy

evidente. Como veremos en la parte práctica, hay veces que vale la pena ser menos estrictos con la atomicidad del test, para evitar abusar de los dobles de prueba.

Independiente significa que un test no puede depender de otros para producir un resultado satisfactorio. No puede ser parte de una secuencia de tests que se deba ejecutar en un determinado orden. Debe funcionar siempre igual independientemente de que se ejecuten otros tests o no.

Inocuo significa que no altera el estado del sistema. Al ejecutarlo una vez, produce exactamente el mismo resultado que al ejecutarlo veinte veces. No altera la base de datos, ni envía emails ni crea archivos, ni los borra. Es como si no se hubiera ejecutado.

Rápido tiene que ser porque ejecutamos un gran número de tests cada pocos minutos y se ha demostrado que tener que esperar unos cuantos segundos cada rato, resulta muy improductivo. Un sólo test tendría que ejecutarse en una pequeña fracción de segundo. La rapidez es tan importante que Kent Beck ha desarrollado recientemente una herramienta que ejecuta los tests desde el IDE Eclipse mientras escribimos código, para evitar dejar de trabajar en código mientras esperamos por el resultado de la ejecución. Se llama JUnit Max14.

Para conseguir cumplir estos requisitos, un test unitario aísla la parte del SUT que necesita ejercitarse de tal manera que el resto está inactivo durante la ejecución. Hay principalmente dos formas de validar el resultado de la ejecución del test: validación del estado y validación de la interacción, o del comportamiento.

Los desarrolladores utilizamos los tests unitarios para asegurarnos de que el código funciona como esperamos que funcione, al igual que el cliente usa los tests de cliente para asegurarse que los requisitos de negocio se alcancen como se espera que lo hagan.

1.1.1.2.5. Pruebas de Integración.

Por último, los tests de integración son la pieza del rompecabezas que nos faltaba para cubrir el vacío entre los tests unitarios y los de sistema. Los tests de integración se pueden ver como tests de sistema pequeños. Típicamente, también se escriben usando herramientas xUnit y tienen un aspecto parecido a los tests unitarios, sólo que estos pueden romper las reglas. Como su nombre indica, integración significa que ayuda a unir distintas partes del sistema. Un test de integración puede escribir y leer de base de datos para comprobar que, efectivamente, la lógica de negocio entiende datos reales. Es el complemento a los tests unitarios, donde habíamos “falseado” el acceso a datos para limitarnos a trabajar con la lógica de manera aislada. Un test de integración podría ser aquel que ejecuta la capa de negocio y después consulta la base de datos para afirmar que todo el proceso, desde negocio hacia abajo, fue bien. Son, por tanto, de granularidad más gruesa y más frágiles

que los tests unitarios, con lo que el número de tests de integración tiende a ser menor que el número de tests unitarios. Una vez que se ha probado que dos módulos, objetos o capas se integran bien, no es necesario repetir el test para otra variante de la lógica de negocio; para eso habrán varios tests unitarios. Aunque los tests de integración pueden saltarse las reglas, por motivos de productividad es conveniente que traten de ser inocuos y rápidos. Si tiene que acceder a base de datos, es conveniente que luego la deje como estaba.

1.1.1. Pruebas unitarias

1.1.1.1. Las pruebas unitarias y su utilidad.

Previamente hemos citado xUnit repetidamente pero xUnit como tal no es ninguna herramienta en sí misma. La letra x es tan sólo un prefijo a modo de comodín para referirnos de manera genérica a los numerosos frameworks basados en el original SUnit. SUnit fue creado por Kent Beck para la plataforma SmallTalk y se ha portado a una gran variedad de lenguajes y plataformas como Java (JUnit), .Net (NUnit), Python (PyUnit), Ruby (RubyUnit), Perl (PerlUnit), C++ (CppUnit), etc. Si aprendemos a trabajar con NUnit o JUnit sabremos hacerlo con cualquier otro framework tipo xUnit porque la filosofía es siempre la misma.

Un test tiene tres partes, que se identifican con las siglas AAA en inglés: Arrange (Preparar), Act (Actuar), Assert (Afirmar).

Una parte de la preparación puede estar contenida en el método SetUp, si es común a todos los tests de la clase. Si la etapa de preparación es común a varios tests de la clase pero no a todos, entonces podemos definir otro método o función en la misma clase, que aúne tal código. No le pondremos la etiqueta de test sino que lo invocaremos desde cada punto en que lo necesitemos.

El acto consiste en hacer la llamada al código que queremos probar (SUT) y la afirmación o afirmaciones se hacen sobre el resultado de la ejecución, bien mediante validación del estado o bien mediante validación de la interacción. Se afirma que nuestras expectativas sobre el resultado se cumplen. Si no se cumplen el framework marcará en rojo cada falsa expectativa.

Veamos un ejemplo en lenguaje C# con el framework NUnit.

```
using System;
using NUnit.Framework;
namespace EjemplosNUnit
{
    [TestFixture]
    public class NameNormalizerTests
    {
        [Test]
        public void FirstLetterUpperCase()
        {
            // Arrange
            string name = "pablo rodriguez";
            NameNormalizer normalizer =

```

```
    new NameNormalizer();
    // Act
    string result =
        normalizer.FirstLetterUpperCase(
            name);
    // Assert
    Assert.AreEqual("Pablo Rodriguez", result);
}
}
```

Hemos indicado a NUnit que la clase es un conjunto de tests (test case), utilizando para ello el atributo `TestFixture`.

El nombre que le hemos puesto a la clase describe el conjunto de los tests que va a contener. Debemos utilizar conjuntos de tests distintos para probar grupos de funcionalidad distinta o lo que es lo mismo: no se deben incluir todos los tests de toda la aplicación en un solo conjunto de tests (una sola clase).

En nuestro ejemplo la clase contiene un único test que está marcado con el atributo `Test`. Los tests siempre son de tipo `void` y sin parámetros de entrada. El nombre del test es largo porque es autoexplicativo. Es la mejor forma de documentarlo. Poner un comentario de cabecera al test, es un antipatrón porque vamos a terminar con un gran número de tests y el esfuerzo de mantener todos sus comentarios es muy elevado. De hecho es un error que el comentario no coincida con lo que hace el código y eso pasa cuando modificamos el código después de haber escrito el comentario. No importa que el nombre del método tenga cincuenta letras, no le hace daño a nadie. Si no sabemos cómo resumir lo que hace el test en menos de setenta letras, entonces lo más probable es que tampoco sepamos qué test vamos a escribir, qué misión cumple. Es una forma de detectar un mal diseño, bien del test o bien del SUT. A veces cuando un código requiere documentación es porque no está lo suficientemente claro.

En el cuerpo del test aparecen sus tres partes delimitadas con comentarios. En la práctica nunca delimitamos con comentarios, aquí está escrito meramente con fines docentes. La finalidad del test del ejemplo es comprobar que el método `FirstLetterUpperCase` de la clase `NameNormalizer` es capaz de poner en mayúscula la primera letra de cada palabra en una frase. Es un test de validación de estado porque hacemos la afirmación de que funciona basándonos en el estado de una variable. `Assert` en inglés viene a significar afirmar. La última línea dice: Afirma que la variable `result` es igual a "Pablo Rodriguez". Cuando NUnit ejecute el método, dará positivo si la afirmación es cierta o negativo si no lo es. Al positivo le llamamos luz verde porque es el color que emplea la interfaz gráfica de NUnit o simplemente decimos que el test pasa. Al resultado negativo le llamamos luz roja o bien decimos que el test no pasa.

Imaginemos que el código del SUT ya está implementado y el test da luz verde. Pasemos al siguiente ejemplo recalmando que todavía no estamos practicando TDD, sino simplemente estamos explicando el funcionamiento de un framework xUnit.

```
[Test]
public void SurnameFirst()
{
```

```

    string name = "gonzalo aller";
    NameNormalizer normalizer =
        new NameNormalizer();
    string result =
        normalizer.SurnameFirst(name);
    Assert.AreEqual("aller, gonzalo", result);
}

```

Es otro test de validación de estado que creamos dentro del mismo conjunto de tests porque el SUT es un método de la misma clase que antes. Lo que el test comprueba es que el método `SurnameFirst` es capaz de recibir un nombre completo y devolver el apellido por delante, separado por una coma. Si nos fijamos bien vemos que la declaración de la variable `normalizer` es idéntica en ambos tests. A fin de eliminar código duplicado la movemos hacia el `SetUp`.

El conjunto queda de la siguiente manera:

```

namespace EjemplosNUnit
{
    [TestFixture]
    public class NameNormalizerTests
    {
        NameNormalizer _normalizer;

        [SetUp]
        public void SetUp()
        {
            _normalizer =
                new NameNormalizer();
        }

        [Test]
        public void FirstLetterUpperCase()
        {
            string name = "pablo rodriguez";
            string result =
                _normalizer.FirstLetterUpperCase(
                    name);
            Assert.AreEqual("Pablo Rodriguez", result);
        }

        [Test]
        public void SurnameFirst()
        {
            string name = "gonzalo aller";
            string result =
                _normalizer.SurnameFirst(
                    name);
            Assert.AreEqual("aller, gonzalo", result);
        }
    }
}

```

Antes de cada uno de los dos tests el framework invocará al método `SetUp` recordándonos que cada prueba es independiente de las demás.

Hemos definido `_normalizer` como un miembro privado del conjunto de tests. El guion bajo (underscore) que da comienzo al nombre de la variable, es una regla de estilo que nos ayuda a identificarla rápidamente como variable de la clase en cualquier parte del código. El método `SetUp` crea una nueva instancia de dicha variable asegurándonos que entre la ejecución de un test y la de otro, se destruye y se vuelve a crear, evitando efectos colaterales. Por tanto lo que un test haga con la variable `_normalizer` no afecta a ningún otro. Podríamos haber extraído también la variable `name` de los tests pero como no se usa nada más que para alimentar al SUT y no interviene en la fase de afirmación, lo mejor es liquidarla:

```
namespace EjemplosNUnit
{
    [TestFixture]
    public class NameNormalizerTests
    {
        NameNormalizer _normalizer;

        [SetUp]
        public void SetUp()
        {
            _normalizer =
                new NameNormalizer();
        }

        [Test]
        public void FirstLetterUpperCase()
        {
            string result =
                _normalizer.FirstLetterUpperCase(
                    "pablo rodriguez");
            Assert.AreEqual("Pablo Rodriguez", result);
        }

        [Test]
        public void SurnameFirst()
        {
            string result =
                _normalizer.SurnameFirst(
                    "gonzalo aller");
            Assert.AreEqual("aller, gonzalo", result);
        }
    }
}
```

Así como hemos creado una prueba en NUnit podemos crear una prueba casi para cualquier lenguaje de programación.

Veamos un ejemplo sencillo con PHPUnit.

```
<?php
// respuesta que le pasaremos a la aserción
$variable = 'texto';
// pregunta o aserción que hace la prueba
```

```
// si $variable no es igual texto entonces la prueba no es correcta
// si $variable es igual a texto entonces la prueba pasa
$this->assertEquals('texto', $variable);
?>
```

Y por último otro ejemplo sencillo en JUnit.

```
import junit.framework.Assert;
```

```
import android.test.AndroidTestCase;
```

```
public class SomeTest extends AndroidTestCase {
```

```
    public void testSomething() throws Throwable {
```

```
        Assert.assertTrue(1 + 1 == 2);
```

```
}
```

```
    public void testSomethingElse() throws Throwable {
```

```
        Assert.assertTrue(1 + 1 == 3);
```

```
}
```

```
}
```

1.2. Pruebas Unitarias con el framework JUnit

JUnit es un framework, creado por Erich Gamma y Kent Beck, utilizado para automatizar las pruebas unitarias en el lenguaje Java. Las pruebas en JUnit se realizan por medio de casos de prueba escritos en clases Java.

Una de las grandes utilidades de JUnit es que los casos de prueba quedan definidos para ser ejecutados en cualquier momento, por lo cual es sencillo, luego de realizar modificaciones al programa, comprobar que los cambios no introdujeron nuevos errores. Esto último es conocido como pruebas de regresión. Cabe destacar que para que un método de prueba sea exitoso o no, son usados los métodos “**assert**”, tanto para comparar los valores esperados contra los valores reales, o directamente hacer que el método de prueba falle. En la figura 1.2 se describen los métodos implementados de JUnit.

Método assert de JUnit	Qué comprueba
assertTrue(expresión)	comprueba que la expresión se evalúe a true
assertFalse(expresión)	comprueba que la expresión se evalúe a false
assertEquals(valor esperado,valor real)	comprueba que el valor esperado sea igual al valor real
assertNull(objeto)	comprueba que el objeto sea null
assertNotNull(objeto)	comprueba que el objeto no sea null
assertSame(objeto Esperado,objeto Real)	comprueba que el objeto Esperado y el objeto Real sean el mismo objeto
assertNotSame(objeto Esperado,objeto Real)	comprueba que el objeto Esperado no sea el mismo que el objeto Real
fail()	hace que el test termine con fallo

Figura 1.3. Métodos assert()

1.2.1. Creación y Ejecución de una prueba unitaria simple con JUnit en Netbeans o Eclipse

Para empezar, vamos a ver el ejemplo más simple posible, para que podamos centrarnos en la comprensión de lo que son las pruebas unitarias y de la herramienta para llevarlas a cabo: **JUnit**. El IDE que utilizaremos será Eclipse.

1. Empezaremos creando la clase que queremos probar. Tendrá el siguiente código:

```
public class Suma {  
  
    private int num1;  
  
    private int num2;  
  
    public Suma(int n1, int n2) {  
        num1 = n1;  
        num2 = n2;  
    }  
    public int sumar() {  
        int resultado = num1 + num2;  
        return resultado;  
    }  
}
```

2. Lo siguiente es crear la clase que nos servirá para probar la clase Suma. Queremos saber si la suma se hace correctamente en tres casos: sumando dos números positivos, sumando dos números negativos y sumando un número positivo y un número negativo. El código será el siguiente:

```
public class SumaTest {  
  
    @Test  
    public void sumaPositivos() {  
        System.out.println("Sumando dos números positivos ...");  
        Suma S = new Suma(2, 3);  
        assertTrue(S.sumar() == 5);  
    }  
  
    @Test  
    public void sumaNegativos() {  
        System.out.println("Sumando dos números negativos ...");  
        Suma S = new Suma(-2, -3);  
        assertTrue(S.sumar() == -5);  
    }  
  
    @Test  
    public void sumaPositivoNegativo() {  
        System.out.println("Sumando un número positivo y un  
        número negativo ...");  
        Suma S = new Suma(2, -3);  
        assertTrue(S.sumar() == -1);  
    }  
}
```

3. Activamos la vista "JUnit" en Eclipse. Para ello hay que pulsar en: "Window"; "Show View"; "Other..."; "Java"; "JUnit".

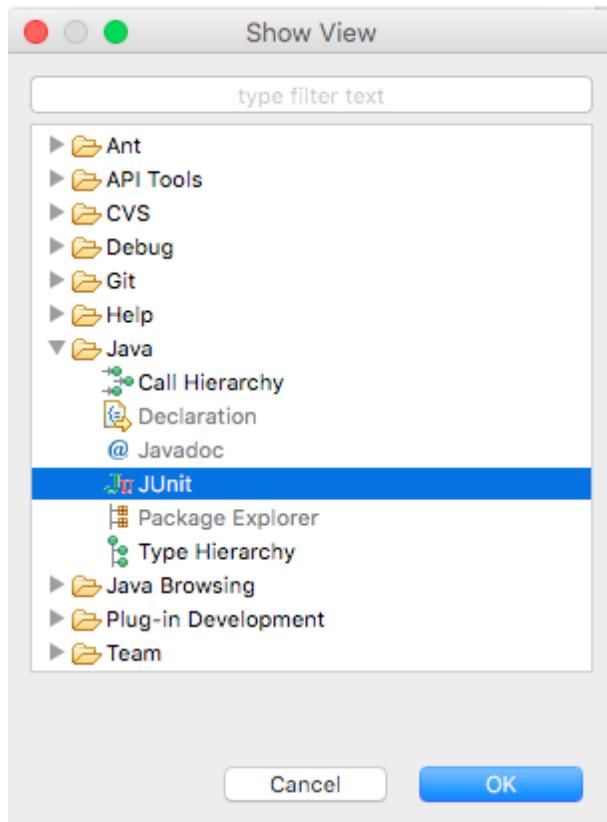


Figura 1.4. Activar la vista JUnit

4. Y ejecutamos el proyecto como test JUnit, pulsando sobre el proyecto con el botón derecho, luego pulsamos "Run as" y finalmente "JUnit Test".

Ahora podremos ver los resultados. Si todo ha ido bien, en la vista JUnit nos aparecerá esto:

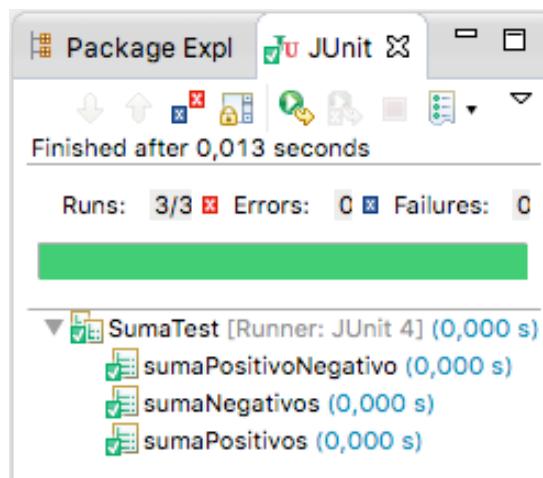


Figura 1.5. Resultados correctos de las pruebas

5. Si ha habido algún fallo nos aparecerá lo siguiente:

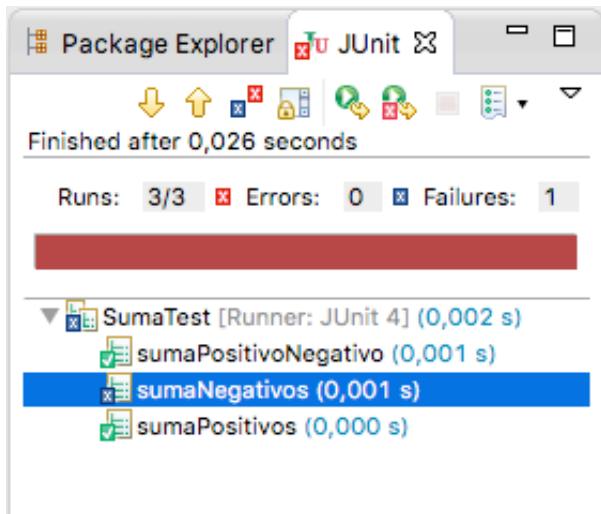


Figura 1.6. Resultados con error en las pruebas

Como vemos, en el primer caso el test ha sido completado sin fallos. Sin embargo, en el segundo caso (se ha modificado el código a propósito para que falle) se ha encontrado un error en “sumaNegativos”.

1.2.2. Creación y Ejecución de una prueba unitaria de regular complejidad con JUnit en Netbeans o Eclipse

Ahora comenzemos a aplicar TDD y para ello enumeraremos los pasos para aplicar TDD:

1) Escribir un test

Debe ser el más sencillo que se nos ocurra - Debe fallar al correrlo

2) Correr todos los tests

Si hay errores, debemos implementar lo mínimo necesario para que pasen y volver al paso 2

3) Reflexiono - ¿Se puede mejorar el código?

- Sí ⇒ Refactorizar y volver al paso 2
- No ⇒ Volver al paso 1

Veamos el ejemplo para determinar si un año es bisiesto bajo las siguientes condiciones:

- Un año es bisiesto si es divisible por cuatro lo que provoca que uno de cada cuatro años sea bisiesto.
- Para un mayor ajuste los años divisibles por 100 no serán bisiestos, de tal forma que cada 100 años habrá un año que debería ser bisiesto y no lo es.
- Sin embargo si el año es divisible por 400 sí que es bisiesto, así cada 400 años habrá un año que no debería ser bisiesto pero sí que lo es.

Paso 1: Escribamos un test

```
public class YearUtilitiesTest {
    @Test
    public void test1() {
        YearUtilities instance = new YearUtilities();
        assertTrue(instance.isLeap(4)); //es bisiesto
```

```

        }
    }
}
```

Paso 2: Correr todos los tests

Nos da error pues el la clase YearUtilities no existe. Implementamos lo mínimo necesario:

```

public class YearUtilities {
    public boolean isLeap(int year){
        return true;
    }
}
```

Paso 3: ¿Se puede mejorar el código?

Si.

¿Realmente necesitamos una instancia de YearUtilities?

```

public class YearUtilitiesTest {
    @Test
    public void test1() {
        assertTrue(YearUtilities.isLeap(4)); //es bisiesto
    }
}
```

Decisión de diseño: Método estático

```

public class YearUtilities {
    public static boolean isLeap(int year){
        return true;
    }
}
```

Paso 1: Escribamos un test

```

@Test
public void test2() {
    assertFalse(YearUtilities.isLeap(6)); //no es bisiesto
}
```

Paso 2: Correr todos los tests

Nos da error pues el el método isLeap devuelve true cuando debería ser false. Implementamos lo mínimo necesario:

```

public class YearUtilities {
    public static boolean isLeap(int year){
        if(year%4 == 0){
            return true;
        }else{
            return false;
        }
    }
}
```

```
}
```

Paso 3: ¿Se puede mejorar el código?

No. Regresamos al Paso 1

Paso 1: Escribamos un test

```
@Test  
public void test3() {  
    assertFalse(YearUtilities.isLeap(100));//no es bisiesto  
}
```

Paso 2: Correr todos los tests

Nos da error pues el el método isLeap devuelve true cuando debería ser false.
Implementamos lo mínimo necesario:

```
public class YearUtilities {  
    public static boolean isLeap(int year){  
        if(year%4 == 0){  
            if(year%100 ==0){  
                return false;  
            }else{  
                return true;  
            }  
        }else{  
            return false;  
        }  
    }  
}
```

Paso 3: ¿Se puede mejorar el código?

No. Regresamos al Paso 1

Paso 1: Escribamos un test

```
@Test  
public void test4() {  
    assertTrue(YearUtilities.isLeap(400));//es bisiesto  
}
```

Paso 2: Correr todos los tests

Nos da error pues el el método isLeap devuelve false cuando debería ser true.
Implementamos lo mínimo necesario:

```
public class YearUtilities {  
    public static boolean isLeap(int year){  
        if(year%4 == 0){  
            if(year%100 ==0){  
                if(year%400 ==0){  
                    return true;  
                }else{  

```

```
        return false;
    }
}else{
    return true;
}
}else{
    return false;
}
}
```

Finalmente podemos agregar otros tests que deben pasar sin problemas y nuestro código Java de tests quedaría como sigue:

```
public class YearUtilitiesTest {

    @Test
    public void test1() {
        assertTrue(YearUtilities.isLeap(4)); //es bisiesto
    }
    @Test
    public void test2() {
        assertFalse(YearUtilities.isLeap(6)); //no es bisiesto
    }
    @Test
    public void test3() {
        assertFalse(YearUtilities.isLeap(100)); //no es bisiesto
    }
    @Test
    public void test4() {
        assertTrue(YearUtilities.isLeap(400)); //es bisiesto
    }
    @Test
    public void test5() {
        assertFalse(YearUtilities.isLeap(2015)); //no es bisiesto
    }
    @Test
    public void test6() {
        assertTrue(YearUtilities.isLeap(2016)); //es bisiesto
    }
}
```



Figura 1.7. Resultados satisfactorios de las pruebas

1.2.3. Creación y Ejecución de una prueba unitaria avanzada con JUnit en Netbeans o Eclipse

Veamos otro ejemplo en que debemos determinar si una contraseña ingresada al sistema cumple los lineamientos de PCI (normas de seguridad de la industria de Tarjetas de Crédito) que dice que debe tener una longitud mínima de siete caracteres y debe ser una combinación de caracteres numéricos y alfabéticos.

Adicionalmente por restricción de nuestra aplicación debemos considerar que la contraseña debe ser de máximo diez caracteres y para darle mayor consistencia debe contener al menos una letra mayúscula.

Paso 1: Escribamos un test

```
public class TestPasswordValidator {  
    @Test  
    public void testValidLength() {  
        PasswordValidator pv = new PasswordValidator();  
        assertEquals(true, pv.isValid("Abcd123"));  
    }  
}
```

Paso 2: Correr todos los tests

Nos da error pues el la clase PasswordValidator no existe. Implementamos lo “mínimo necesario”:

```
public class PasswordValidator {  
    public boolean isValid(String password) {  
        if (password.length() >= 7 && password.length() <= 10) {  
            return true;  
        }  
        else {  
            return false;  
        }  
    }  
}
```

Paso 3: ¿Se puede mejorar el código?

Si.

¿Realmente necesitamos una instancia de PasswordValidator?

```
public class TestPasswordValidator {  
    @Test  
    public void testValidLength() {  
        assertEquals(true, PasswordValidator.isValid("Abcd123"));  
    }  
}
```

Decisión de diseño: Método estático

```
public class PasswordValidator {  
    public static boolean isValid(String password) {  
        if (password.length() >= 7 && password.length() <= 10) {  
            return true;  
        }  
        else {  
            return false;  
        }  
    }  
}
```

¿Puede ser aún más simple?

```
public class PasswordValidator {  
    public static boolean isValid(String password) {  
        return (password.length() >= 7 &&  
                password.length() <= 10);  
    }  
}
```

Mantener constantes como el 7 y el 10 dentro del código puede ser peligroso

```
public class PasswordValidator {  
    private final static int MIN_PW_LENGTH = 7;  
    private final static int MAX_PW_LENGTH = 10;  
    public static boolean isValid(String password) {  
        return (password.length() >= MIN_PW_LENGTH &&
```

```
        password.length() <= MAX_PW_LENGTH);  
    }  
}
```

Paso 1: Escribamos un test

```
@Test  
public void testNotDigit() {  
    assertEquals(false, PasswordValidator.isValid("Abcdefg"));  
}
```

Paso 2: Correr todos los tests

Nos da error pues el el método isValid devuelve true cuando debería ser false.
Implementamos lo mínimo necesario:

```
import java.util.regex.Pattern;  
  
public class PasswordValidator {  
    private final static int MIN_PW_LENGTH = 7;  
    private final static int MAX_PW_LENGTH = 10;  
    public static boolean isValid(String password) {  
        return (password.length() >= MIN_PW_LENGTH &&  
                password.length() <= MAX_PW_LENGTH) &&  
                Pattern.matches(".*\\p{Digit}.*", password);  
    }  
}
```

Paso 3: ¿Se puede mejorar el código?

No. Regresamos al Paso 1

Paso 1: Escribamos un test

```
@Test  
public void testNotUpperCase() {  
    assertEquals(false, PasswordValidator.isValid("abcdef1"));  
}
```

Paso 2: Correr todos los tests

Nos da error pues el el método isValid devuelve true cuando debería ser false.
Implementamos lo mínimo necesario:

```
import java.util.regex.Pattern;  
  
public class PasswordValidator {  
    private final static int MIN_PW_LENGTH = 7;  
    private final static int MAX_PW_LENGTH = 10;  
    public static boolean isValid(String password) {  
        return (password.length() >= MIN_PW_LENGTH &&  
                password.length() <= MAX_PW_LENGTH) &&
```

```
        Pattern.matches(".*\|p{Digit}.*", password) &&
        password.toLowerCase() != password;
    }
}
```

Finalmente podemos agregar otros tests que deben pasar sin problemas y nuestro código Java de tests quedaría como sigue:

```
public class TestPasswordValidator {
    @Test
    public void testValidLength() {
        assertEquals(true, PasswordValidator.isValid("Abcd123"));
    }
    @Test
    public void testNotDigit() {
        assertEquals(false, PasswordValidator.isValid("Abcdefg"));
    }
    @Test
    public void testNotUpperCase() {
        assertEquals(false, PasswordValidator.isValid("abcdef1"));
    }
    @Test
    public void testUpperCaseDigit() {
        assertEquals(true, PasswordValidator.isValid("AAbcdeee4"));
    }
    @Test
    public void testUpperCaseNotDigit() {
        assertEquals(false, PasswordValidator.isValid("AAbcdeefes"));
    }
}
```

Actividades

CASO 1: NÚMEROS ROMANOS

Implemente los casos de prueba necesarios en JUnit e implemente una clase siguiendo la técnica TDD con un método que teniendo como entrada un número arábigo devuelva el mismo en formato de número romano.

CASO 2: CASO DE USO SALARIO DE VENDEDOR

A continuación, se muestra las reglas de negocio para el cálculo del salario de los empleados de una tienda comercial. Supongamos que tenemos la clase EmpleadoCA:

Método	Especificación
float calculaSalarioBruto(TipoEmpleado tipo, float ventasMes, float horasExtra)	El salario base será S/ 1000 si el empleado es de tipo “vendedor”, y de S/ 1500 si es de “encargado”. A esta cantidad se le sumará una prima de S/ 100 si ventasMes es mayor o igual que S/ 1000, y de S/ 200 si fuese al menos de S/ 1500. Por último, cada hora extra se pagará a S/ 20. Si tipo es null, o ventasMes o horasExtra toman valores negativos el método lanzará una excepción de tipo CAException.
float calculaSalarioNeto(float salarioBruto)	Si el salario bruto es menor de S/ 1000, no se aplicará ninguna retención. Para salarios a partir de S/ 1000, y menores de S/ 1500 se les aplicará un 16%, y a los salarios a partir de S/ 1500 se les aplicará un 18%. El método nos devolverá salarioBruto * (1-retencion), o CAExcepcion si el salario es menor que cero.

A partir de dichas especificaciones podemos diseñar un conjunto de casos de prueba siguiendo métodos como el método de pruebas de particiones. Supondremos que después de aplicar un método de pruebas hemos obtenido los siguientes casos de prueba:

Método a probar	Entrada	Salida esperada
calculaSalarioBruto	vendedor, S/ 2000, 8h	1360
calculaSalarioBruto	vendedor, S/ 1500, 3h	1260
calculaSalarioBruto	vendedor, S/ 1499.99, 0h	1100
calculaSalarioBruto	encargado, S/ 1250, 8h	1760
calculaSalarioBruto	encargado, S/ 1000, 0h	1600
calculaSalarioBruto	encargado, S/ 999.99, 3h	1560
calculaSalarioBruto	encargado, S/ 500, 0h	1500
calculaSalarioBruto	encargado, 0 soles, 8h	1660
calculaSalarioBruto	vendedor, -1 soles, 8h	CAExcepcion
calculaSalarioBruto	vendedor, S/ 1500, -1h	CAExcepcion
calculaSalarioBruto	null, S/ 1500, 8h	CAExcepcion
calculaSalarioNeto	2000	1640
calculaSalarioNeto	1500	1230
calculaSalarioNeto	1499.99	1259.9916
calculaSalarioNeto	1250	1050
calculaSalarioNeto	1000	840
calculaSalarioNeto	999.99	999.99
calculaSalarioNeto	500	500
calculaSalarioNeto	0	0
calculaSalarioNeto	-1	CAExcepcion

Implemente los casos de prueba en JUnit e implemente la clase EmpleadoCA siguiendo la técnica TDD.

Resumen

- ❑ **TDD** es una Técnica de Desarrollo basada en “testear” primero:
 - Es más que una técnica de testing
 - Es más que una técnica de programación
 - Es más que una técnica de diseño
 - Es todo en junto
- ❑ **TDD** surge como una de las técnicas de eXtreme Programming
- ❑ Todo test unitario debe ser: Atómico, Independiente, Inocuo y Rápido
- ❑ Un test tiene tres partes, que se identifican con las siglas AAA en inglés: Arrange (Preparar), Act (Actuar), Assert (Afirmar).
- ❑ **JUnit** es un framework, creado por Erich Gamma y Kent Beck, utilizado para automatizar las pruebas unitarias en el lenguaje Java.
- ❑ Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.
 - ☞ http://www.carlosble.com/downloads/disenoAgilConTdd_ebook.pdf
Aquí encontrará una guía completa de TDD elaborada por Carlos Blé.
 - ☞ <http://es.slideshare.net/FernandoPrez16/introduccion-a-tdd-y-unit-testing>
Aquí encontrará una introducción didáctica a TDD.



FUNDAMENTOS RATIONAL FUNCTIONAL TESTER

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, el alumno crea y agrega características avanzadas a los scripts de pruebas funcionales a partir de los casos de prueba de una aplicación Java.

TEMARIO

2.1. Tema 3: Introducción al Rational Functional Tester

- 2.1.1. Arquitectura de Rational Functional Tester
- 2.1.2. Configuración del entorno de pruebas
- 2.1.3. Configuración de aplicaciones Java a probar
- 2.1.4. Proyectos de pruebas funcionales en Rational Functional Tester

2.2. Tema 2: Script de pruebas funcionales

- 2.2.1. Grabación de un script
- 2.2.2. Reproducción de un script
- 2.2.3. Revisión de los resultados
- 2.2.4. Características avanzadas de script de pruebas

ACTIVIDADES PROPUESTAS

- Los alumnos crean script de pruebas funcionales para una aplicación de escritorio Java.
- Los alumnos crean script de pruebas funcionales para una aplicación web Java.
- Los alumnos agregan características avanzadas a los script de pruebas funcionales para una aplicación web Java.

2.1. INTRODUCCIÓN AL RATIONAL FUNCTIONAL TESTER

IBM *Rational Functional Tester* (RFT) es una herramienta automatizada para la realización de pruebas funcionales y de regresión. Permite probar aplicaciones Java, .NET y basadas en Web.

Con la **tecnología de grabación**, se puede generar scripts mediante la ejecución y el uso de la aplicación bajo prueba. Los scripts del RFT son implementados como programas Java. Asimismo, crea una vista que muestra las acciones del usuario sobre la aplicación.

Durante el proceso de grabación, se puede añadir **comandos controlados por datos** que permitirán probar, durante el proceso de reproducción, otros datos provenientes de un **pool de datos**. Otra característica importante del RFT que pueden ser añadidos en un script, son los **puntos de verificación** que permiten evaluar los datos y propiedades de los objetos de la aplicación y confirmar el estado de dichos objetos probándolas aún en versiones posteriores. Esto es posible gracias a la **tecnología ScriptAssure** que permite crear scripts de prueba más resistentes a los cambios en los objetos de las aplicaciones. Los objetos que son referenciados por las aplicaciones son almacenados automáticamente en un **mapa de objetos de prueba** al crear el script.

La siguiente figura muestra la perspectiva **Functional Test** que se utilizará en el RFT:

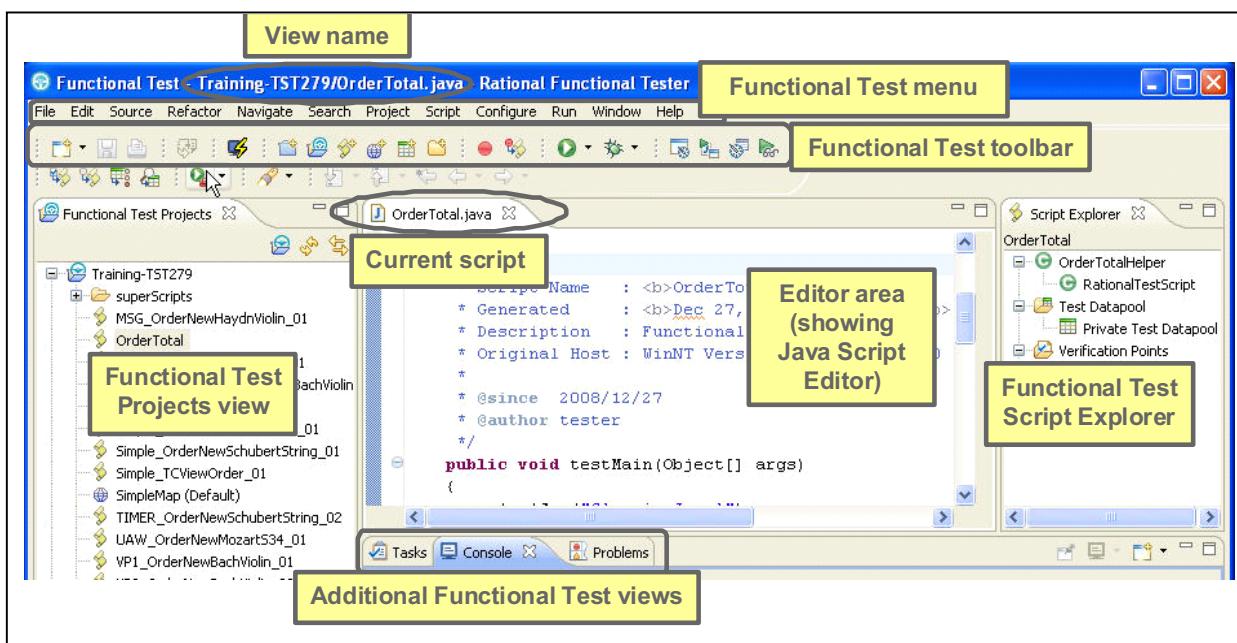


Figura 2.1. Perspectiva *Functional Test*

2.1.1. Arquitectura de Rational Functional Tester

Este apartado introduce una visión general de la arquitectura del RFT el cual es descrito con más detalle en temas posteriores.

2.1.1.1. Almacenamiento de activos de pruebas.

RFT es una herramienta de escritorio basada en archivos y no necesita trabajar con un servidor de componentes. Los activos de pruebas que se almacenan en el RFT son principalmente archivos de

Java y XML. Se puede utilizar una herramienta de gestión de configuración como *Rational Team Concert*, *Rational Clear Case* o herramientas como CVS para el control individual de versiones de los activos de pruebas.

Si se utiliza RFT *Java Scripting*, los proyectos de pruebas funcionales son una clase de proyectos Java de Eclipse™. Si se utiliza RFT *Visual Basic .NET Scripting*, los proyectos de pruebas funcionales son una clase de proyectos de Visual Basic de Microsoft Visual Studio®. Ambos son proyectos creados en el entorno del RFT.

2.1.1.2. Almacenamiento de resultados de pruebas.

Los resultados son almacenados en archivos conocidos como *test logs* dentro del IBM RFT, los cuales pueden ser grabados en muchos formatos diferentes que pueden ser configurados, inclusive en formatos que podrán ser utilizados en herramientas de gestión de pruebas tales como *Rational Quality Manager*. Los formatos que podrá configurarse en el IBM RFT para los *test logs* son los siguientes:

- HTML
- Texto
- Plataforma de herramientas de pruebas y rendimiento
- XML
- *Rational Quality Manager*

2.1.1.3. Grabación de pruebas funcionales.

Es probable que utilice las opciones de grabación del RFT para crear nuevos scripts de prueba, pues es la manera más rápida y sencilla de generar líneas de código, aún si los scripts son modificados posteriormente. Durante el proceso de grabación debe asegurarse de no realizar acciones que no deben ser parte de las pruebas.

Los scripts de pruebas contienen información adicional aparte del archivo que contiene las acciones del usuario interactuando con el aplicativo bajo prueba y del script generado con código Java. La información adicional es conocida como activos de pruebas: objetos de pruebas, puntos de verificación y datos de prueba.

2.1.1.4. Ejecución de pruebas.

Las pruebas grabadas son ejecutadas en modo de reproducción. En este modo, el RFT envía todas las acciones del mouse y el teclado que se graban a la aplicación bajo prueba. En general, no se debe manipular el teclado o el mouse cuando el RFT está en modo de reproducción. Sin embargo, a veces, puede ejecutar las pruebas en modo interactivo para manipular los posibles problemas de reproducción.

Un script de prueba se compone en gran parte de instrucciones de interacción, incluyendo la realización de pruebas con varios objetos de la aplicación bajo prueba. Cuando se ejecuta un script de prueba, RFT primero tiene que identificar y encontrar cada objeto, haciendo coincidir las propiedades de reconocimiento en el mapa de objetos de prueba grabados en el script con los objetos actuales que están

presentes en la aplicación que está en ejecución. Si RFT no puede encontrar un objeto que coincida con el que se encuentra en el script grabado, se registra un error con los intentos de continuar o se anula. En cambio, si RFT encuentra un objeto que coincide con las propiedades grabadas en el script, se realiza la acción sobre el objeto. Estas acciones podrían ser las interacciones del usuario, como hacer un clic, hacer una selección, u otras operaciones, tales como obtener o establecer valores sobre el objeto. Por último, las acciones realizadas en el objeto podría ser un punto de verificación, en cuyo caso RFT compara el valor esperado o un intervalo de valores con el resultado real obtenido en tiempo de ejecución.

Puede reproducir una prueba en la misma máquina o en cualquier otra máquina que ejecuta *Rational Agent*, que se instala por defecto con RFT. También puede ejecutar varias pruebas en varios equipos remotos para pruebas funcionales distribuidos. Esto hace que sea posible realizar muchas más pruebas en un corto periodo de tiempo. Una máquina dada sólo puede ejecutar una prueba a la vez, o muchas pruebas en forma secuencial. También, puede ejecutar las pruebas de RFT en máquinas remotas usando las herramientas de administración de pruebas, tales como *Rational Quality Manager*.

2.1.1.5. Integración con otras aplicaciones.

RFT es un producto *stand alone* que no requiere de otras herramientas o aplicaciones. Sin embargo, de modo opcional, puede integrarse con otros tipos de aplicaciones según las necesidades:

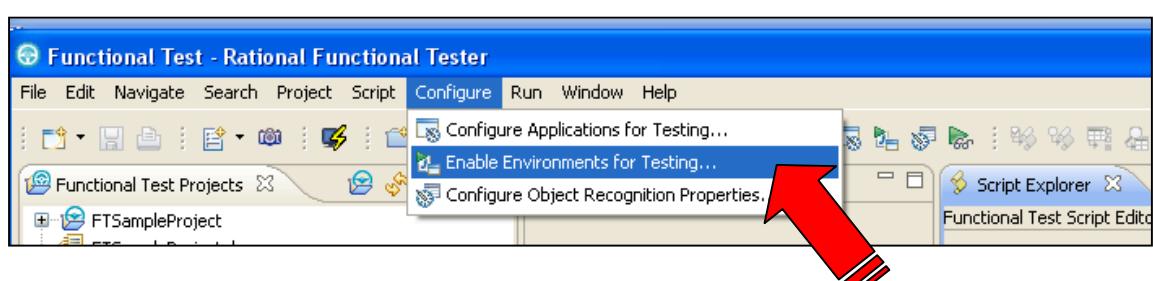
- Para gestión de pruebas o de calidad con *IBM Rational Quality Manager*
- Para gestión de solicitudes de cambios o seguimiento de defectos, *IBM Rational ClearQuest*
- Para gestión de configuraciones o control de versiones, *IBM Rational Team Concert* e *IBM Rational ClearCase*
- Para realizar pruebas unitarias o utilizar herramientas de desarrollo de pruebas, *JUnit*
- Para la generación de pruebas automatizadas, *IBM Rational Performance Tester*
- Herramientas de desarrollo, *IBM Rational Application Developer*

2.1.2. Configuración del entorno de pruebas

Las tareas que debe efectuar para configurar el entorno para pruebas funcionales son:

- Configure los navegadores web.
- Configure los entornos de Java.
- Habilite plataformas de eclipse.

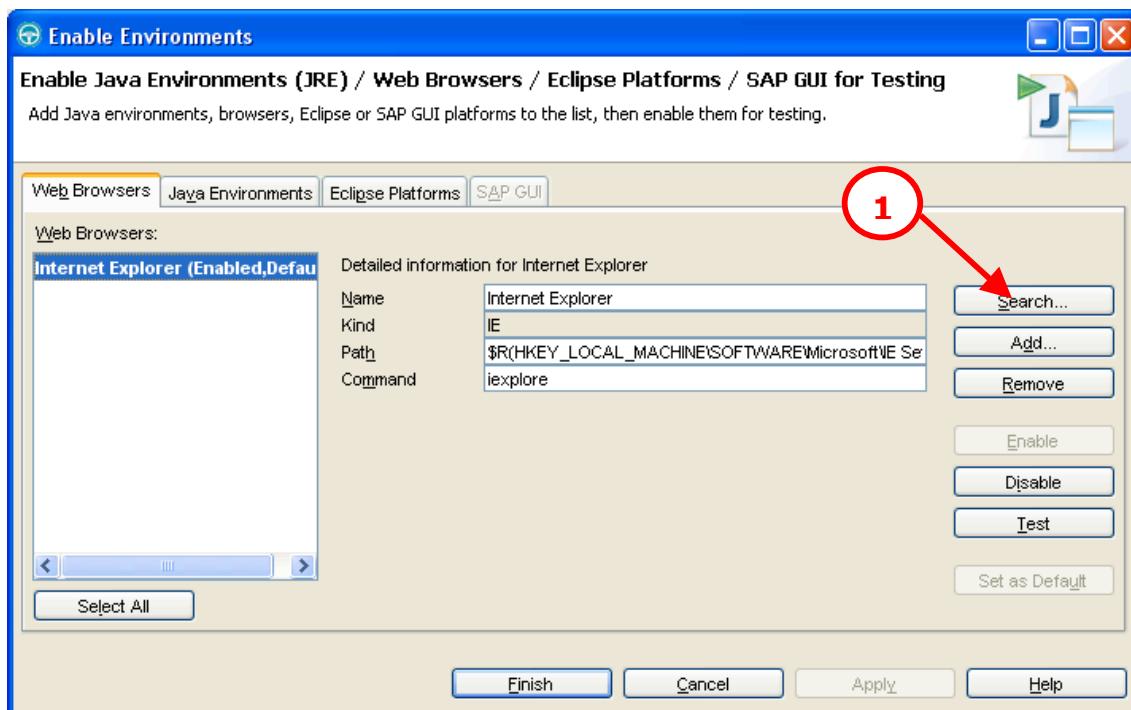
Las tres configuraciones se realiza a partir de la opción ***Configure > Enable Environments for Testing*** desde la barra de menú del RFT.



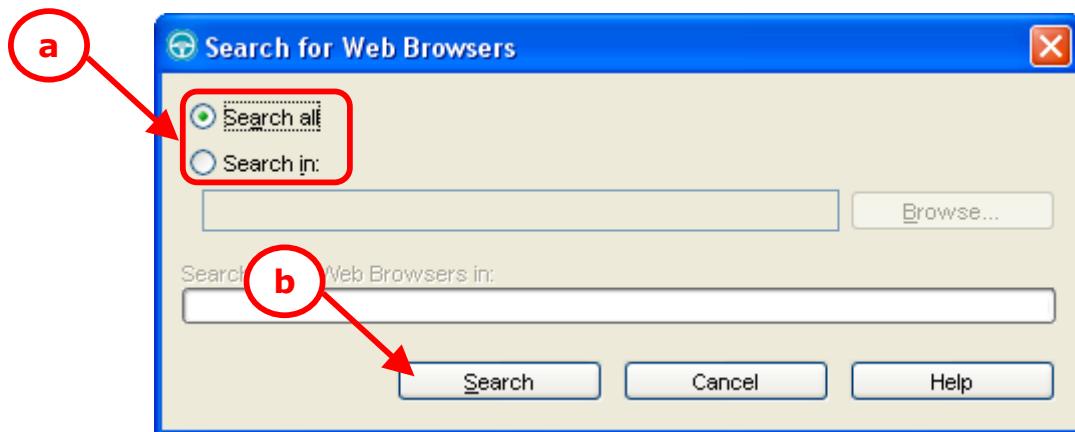
2.1.2.1. Configuración de navegadores web.

Por defecto, al instalar RFT se habilita el navegador web *Internet Explorer*, pero si desea configurar otro navegador que instala posteriormente, debe realizar los pasos que se describen a continuación:

1. Desde la pestaña **Web Browsers** seleccione la opción **Search**.



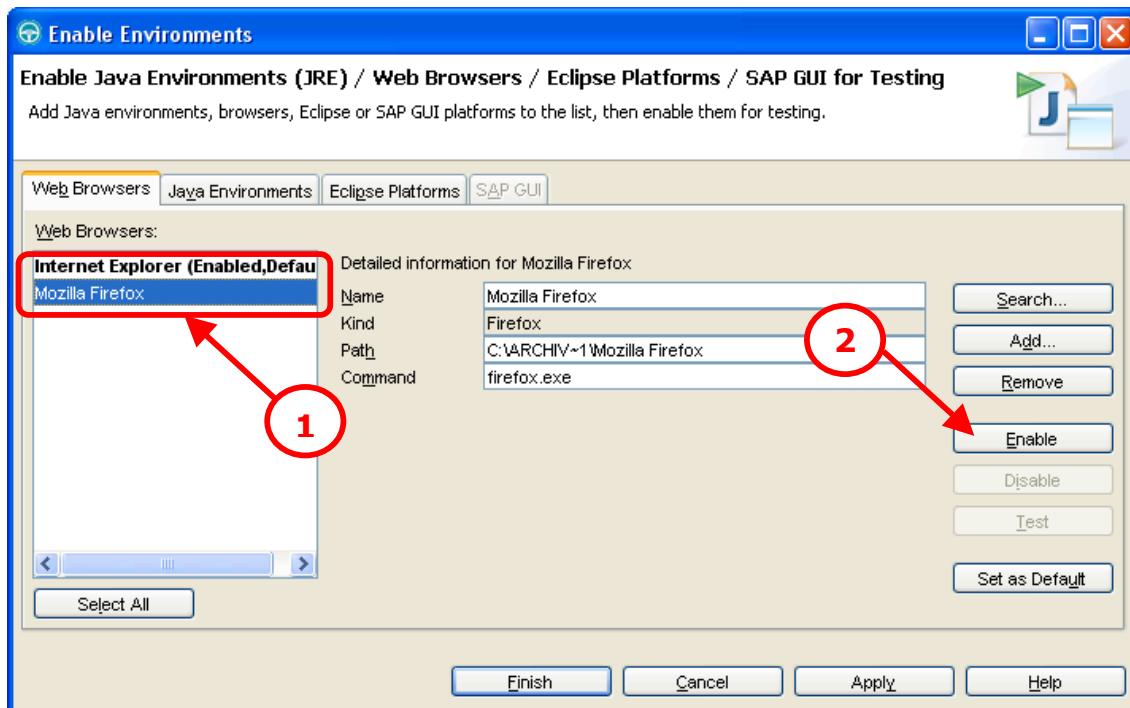
2. A continuación, se abrirá el cuadro de diálogo “Buscar navegadores Web”.



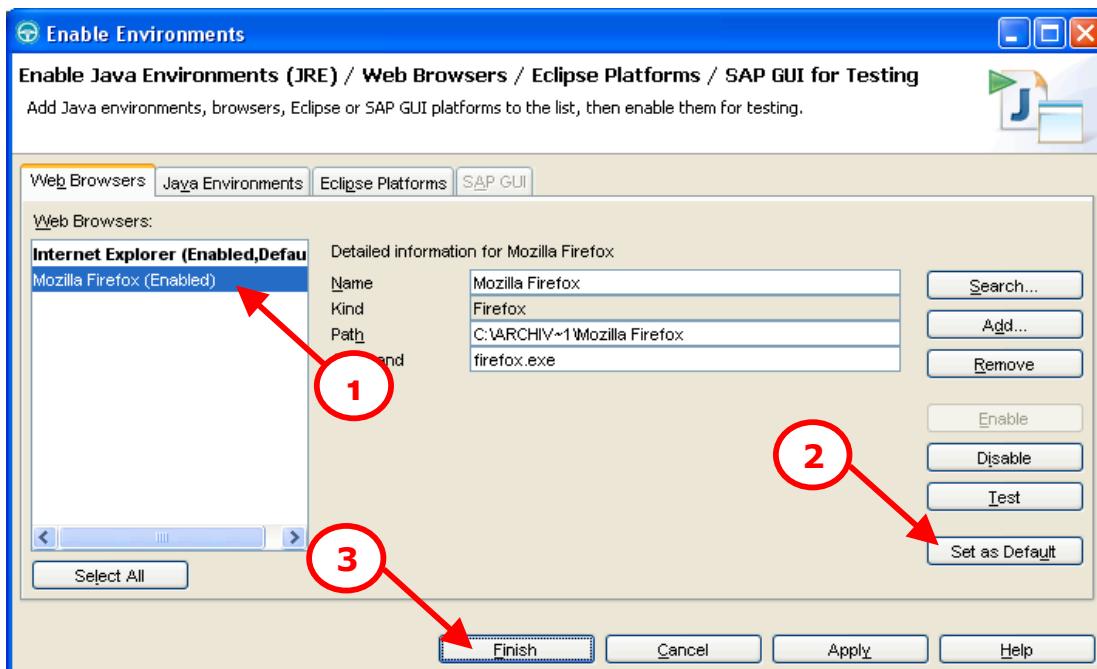
- Selecione uno de los mecanismos de búsqueda siguientes.
 - **Search all** (Buscar todo) para que el habilitador encuentre todos los navegadores del sistema. RFT explorará todos los discos duros o particiones, y enumerará los navegadores en la lista “Navegadores web” de la ventana anterior.
 - **Search in** (Buscar en) para situarse y buscar en una unidad o directorio raíz específicos.

Nota: No debe utilizar la opción “Buscar todo” para buscar navegadores en sistemas Linux o UNIX. En lugar de esta opción, utilice la opción “Buscar en” y búsquelos.

- b. Pulse el botón **Search**.
3. Seleccione los navegadores que desea habilitar de la lista **Web Browsers**. Puede seleccionar varios navegadores mediante la tecla **ctrl** al efectuar la selección o seleccionar todo si desea habilitar todos los navegadores de la lista mediante el botón **Select All**. Luego, pulse **Enable**.



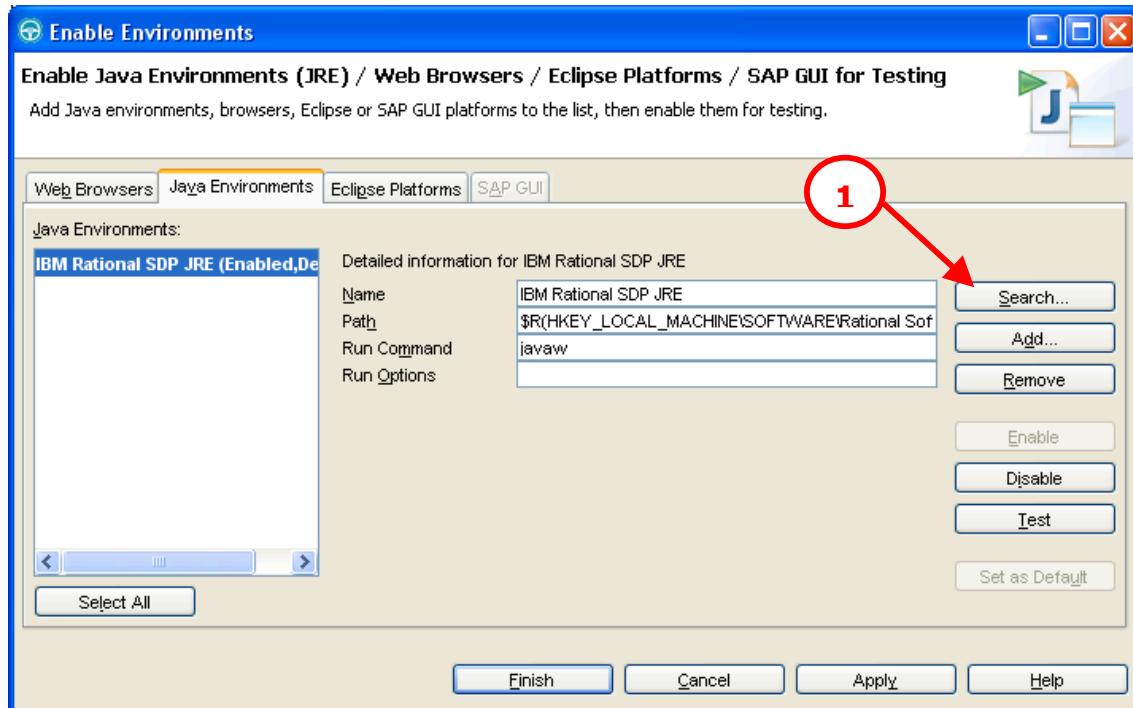
4. Seleccione un navegador como predeterminado y pulse el botón **Set as Default**. Por último, pulse **Apply** para continuar con otra configuración o **Finish** para finalizar.



2.1.2.2. Configuración de entornos Java.

Por defecto, al instalar RFT se habilita un entorno JRE, pero si desea configurar otra versión nueva que instala posteriormente, debe realizar los siguientes pasos:

1. Desde la pestaña **Java Environments** seleccione la opción **Search**.



2. A continuación, se abrirá el cuadro de diálogo “Buscar entornos Java”.

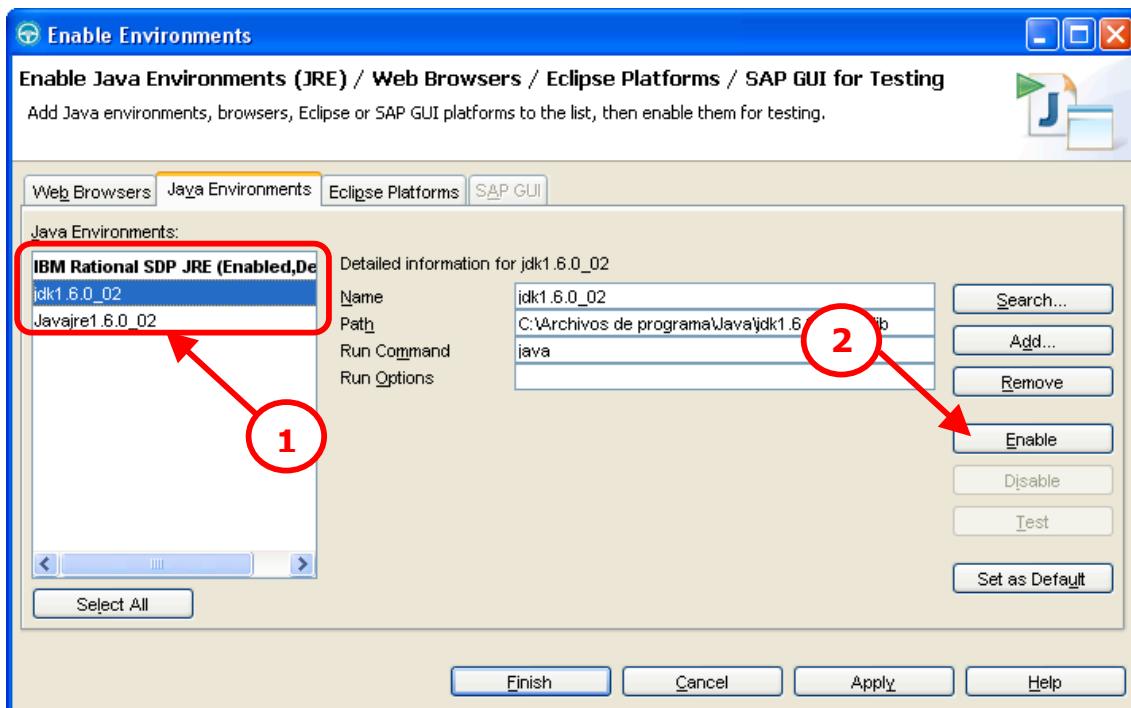


- Selecione uno de los mecanismos de búsqueda siguientes:

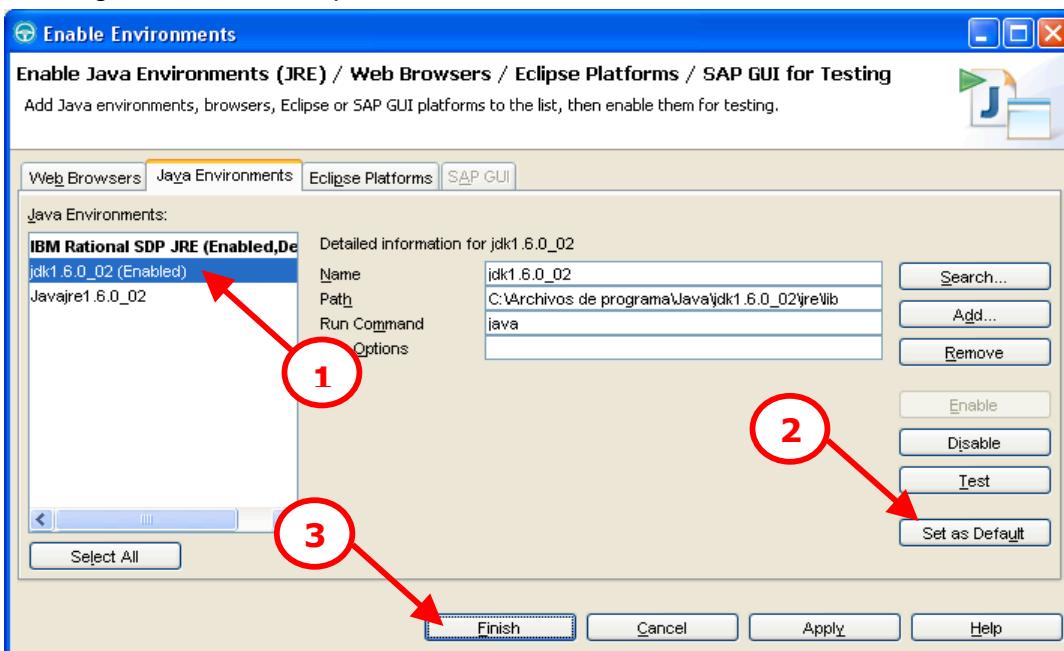
- **Quick search** (Búsqueda rápida), sólo se puede utilizar en sistemas Windows. Éste busca en el registro de Windows todos los entornos de Java y es más rápido que realizar búsquedas en el/los disco(s) duro(s).
- **Search all drives** (Buscar todas las unidades), explora todas las unidades de disco duro o particiones para ubicar todos los entornos de Java del sistema.
- **Search in** (Buscar en) para situarse y buscar en una unidad o directorio raíz específicos.

Nota: No debe utilizar la opción “Buscar todas las unidades” para buscar JRE en sistemas Linux. En lugar de esta opción, utilice la opción Buscar en y busque los JRE.

- b. Despues de elegir uno de los mecanismos de búsqueda, pulse el botón **Search**.
3. Al terminar la búsqueda, RFT lista los JRE en la lista “Entornos Java” de la pestaña **Java Environments**. Ahora, decida qué entorno desea habilitar. Para ello, seleccione uno, o varios JRE mediante la tecla **Ctrl** al efectuar la selección o pulsar **Select All** si desea habilitar todos los JRE. Luego, pulse **Enable**.



4. A continuación, seleccione un JRE para que sea el valor predeterminado y pulse el botón **Set as Default**. Por último, pulse **Apply** para continuar con otra configuración o **Finish** para finalizar.

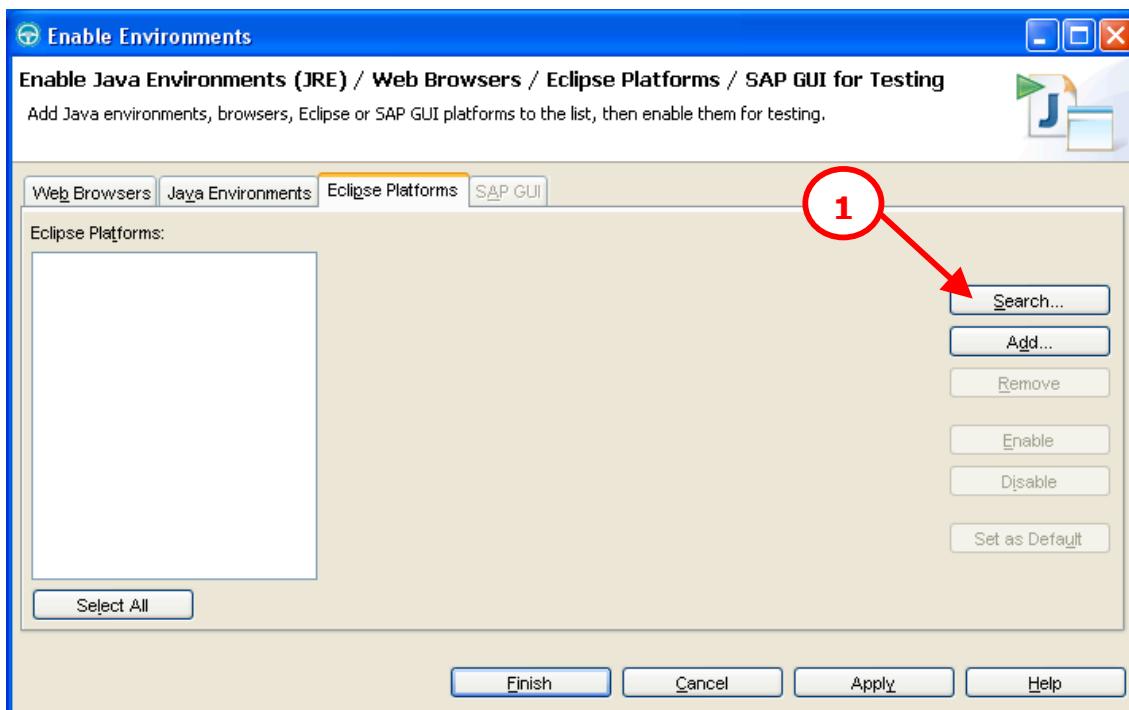


2.1.2.3. Habilitación de plataformas de eclipse.

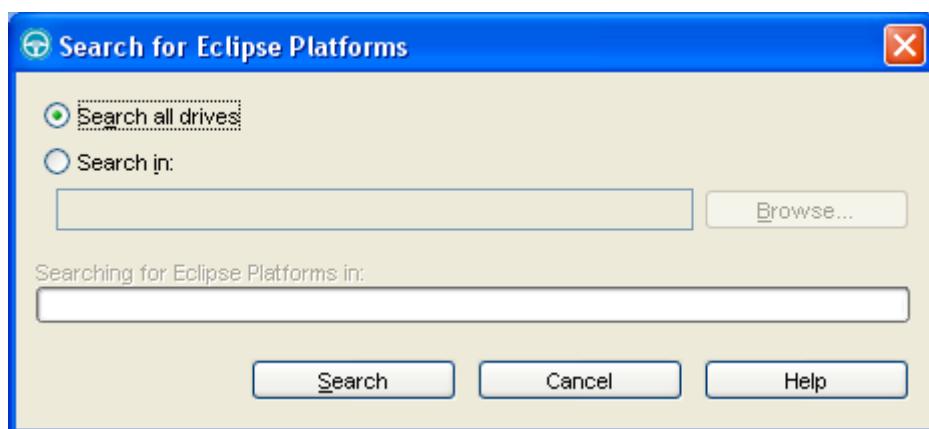
Para habilitar una plataforma de eclipse realice los siguientes pasos:

1. Desde la pestaña **Eclipse Platforms** seleccione la opción **Search**.

Nota: Utilice el botón **Add** para ubicar una plataforma de Eclipse y añadirla directamente.

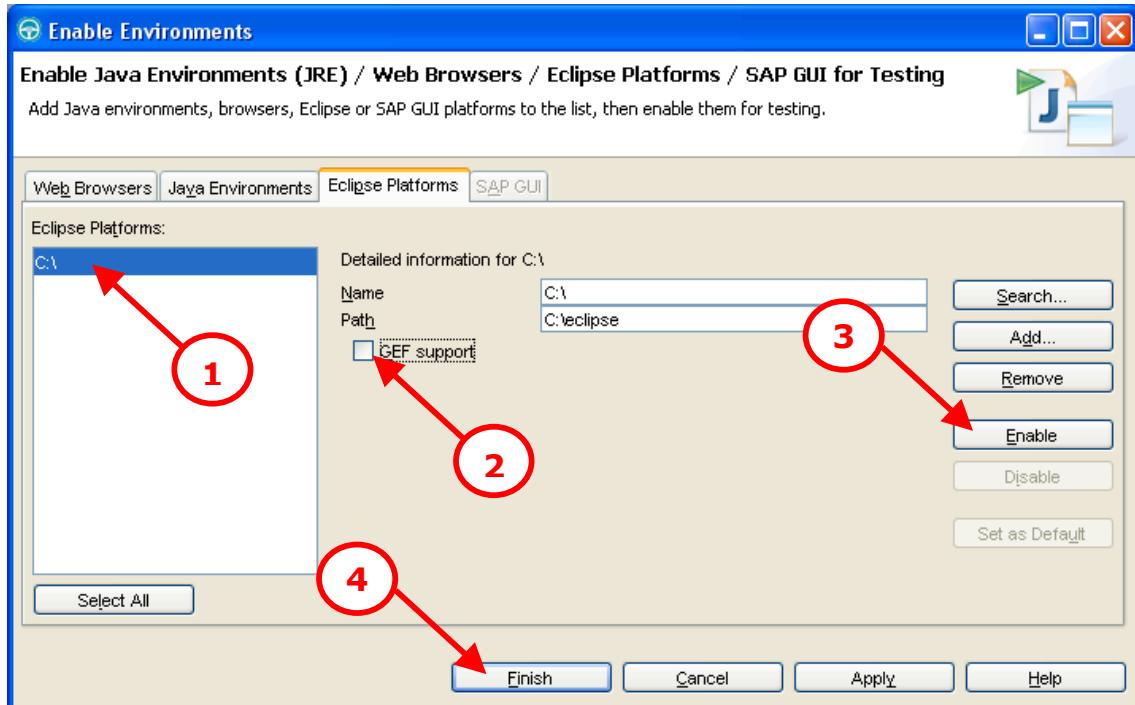


2. De inmediato, se abrirá el cuadro de diálogo “Buscar plataformas de Eclipse”.
 - a. Seleccione uno de los mecanismos de búsqueda siguientes:
 - **Search all drives**, busca las plataformas de Eclipse en todas las unidades de disco.
 - **Search in**, busca la plataforma de Eclipse en un directorio específico.



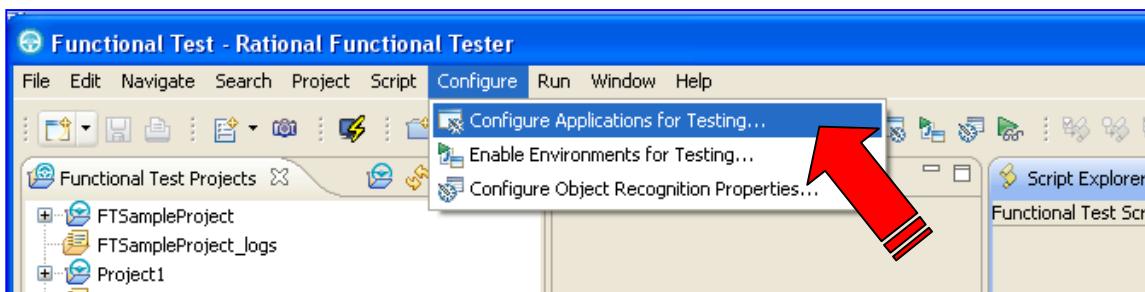
- a. Después de elegir uno de los mecanismos de búsqueda, pulse el botón **Search**.

3. Los resultados de la búsqueda aparecen listados en el panel izquierdo de la pestaña **Eclipse Platforms**. Ahora, seleccione la plataforma de Eclipse que desea habilitar. Luego, para habilitar el soporte para GEF, marque el recuadro de selección Soporte para GEF (el plug-in de habilitación de GEF se copia en el directorio de plug-ins de AUT). A continuación, pulse **Enable**. Por último, pulse **Apply** para continuar con otra configuración o **Finish** para finalizar.



2.1.3. Configuración de aplicaciones Java a probar

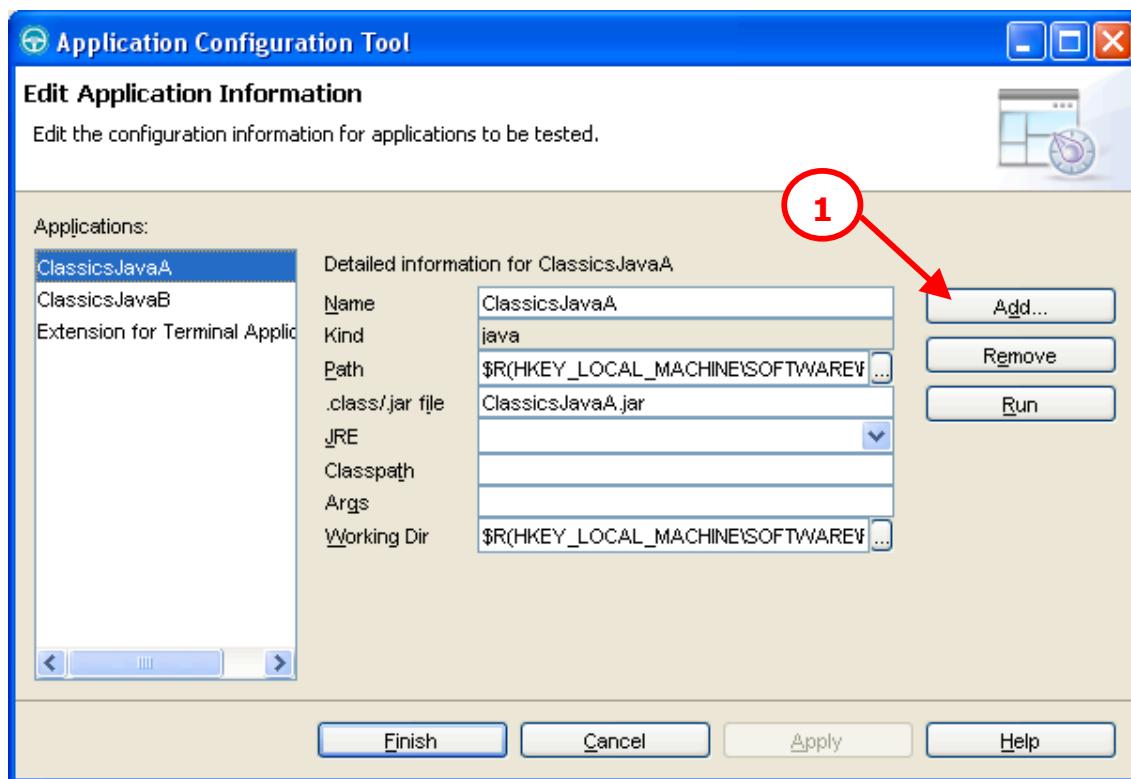
Debe configurar las aplicaciones Java, HTML, VB.NET o Windows para pruebas con RFT a fin de proporcionar el nombre, la vía de acceso y otra información que RFT utiliza para iniciar y ejecutar la aplicación. Para ello, desde la barra de menú principal, seleccione **Configure > Configure Applications for Testing**.



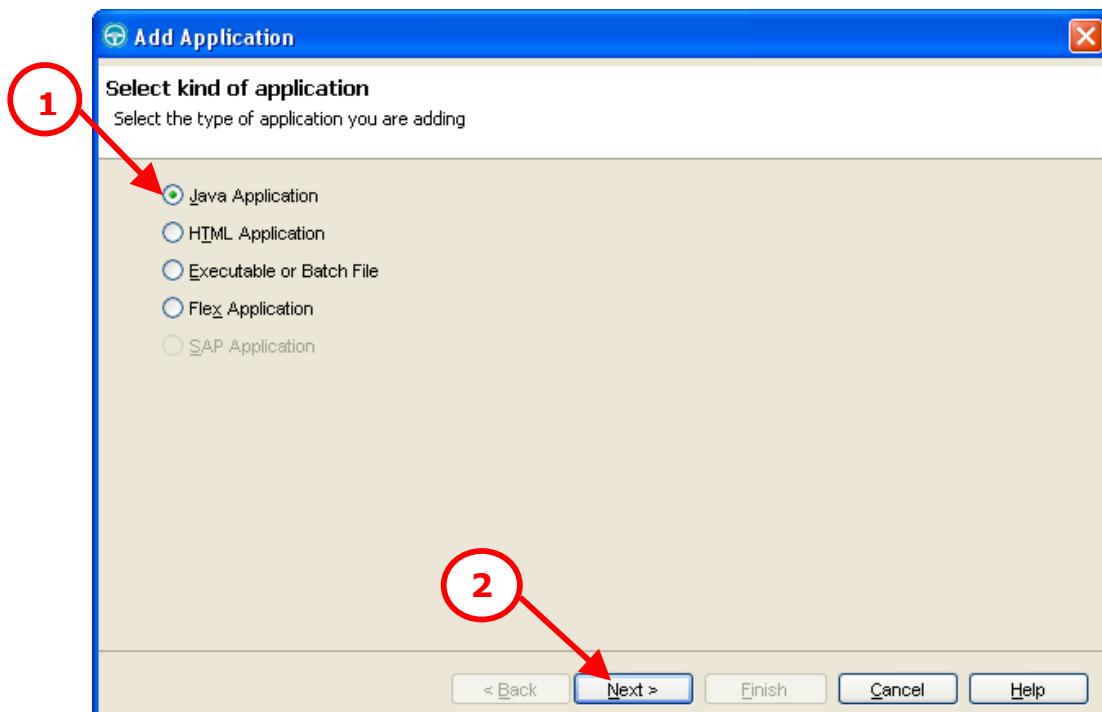
A continuación, como ejercicio agregaremos 2 aplicaciones: una del tipo **Java Application** y otra del tipo **HTML Application**.

2.1.3.1. Agregar una aplicación Java.

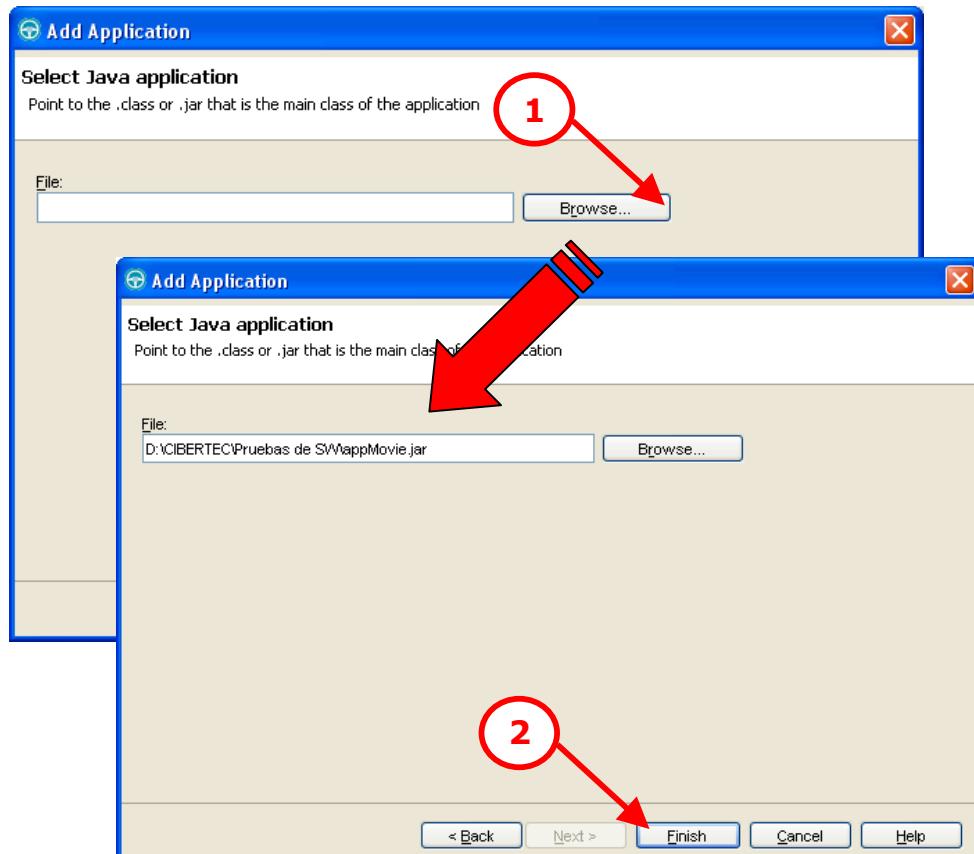
1. Desde la ventana “Herramienta de Configuración de Aplicación” pulse el botón **Add** para añadir una nueva aplicación Java.



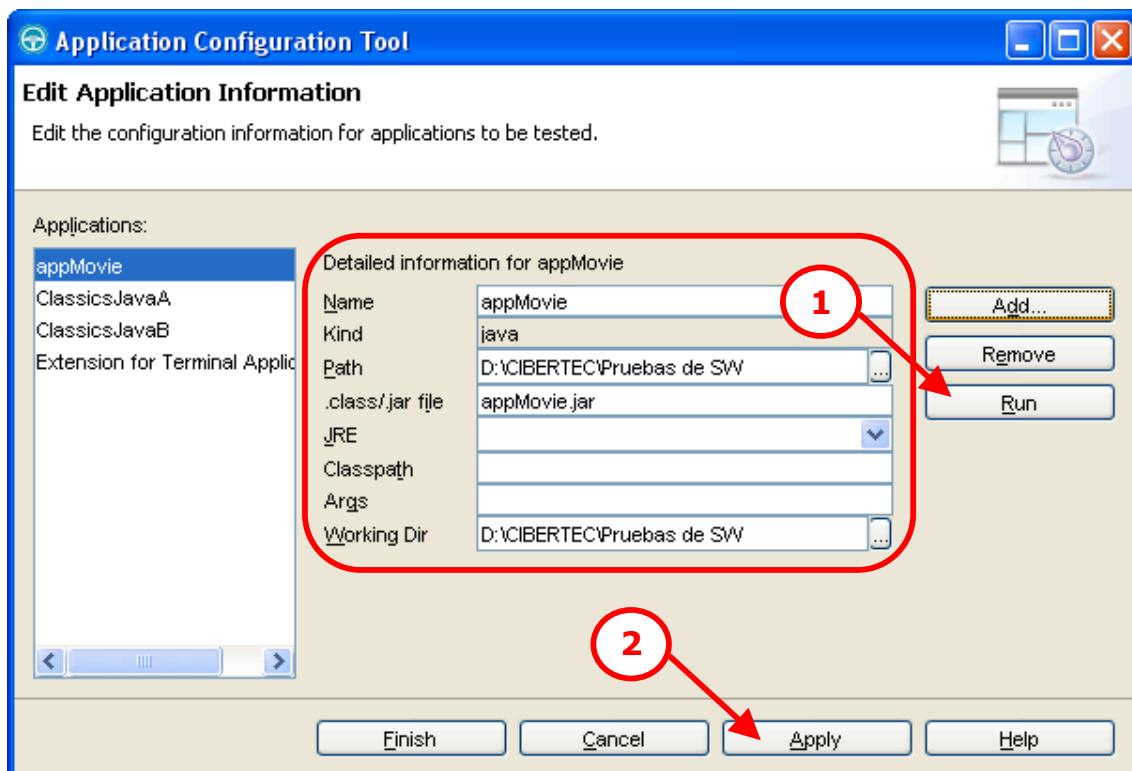
2. En el cuadro de diálogo “Añadir aplicación”, seleccione el tipo de aplicación. En este caso, seleccione **Java Application** y luego pulse **Next**.



3. Pulse **Browse** para ubicar la aplicación y luego pulse **Finish**.

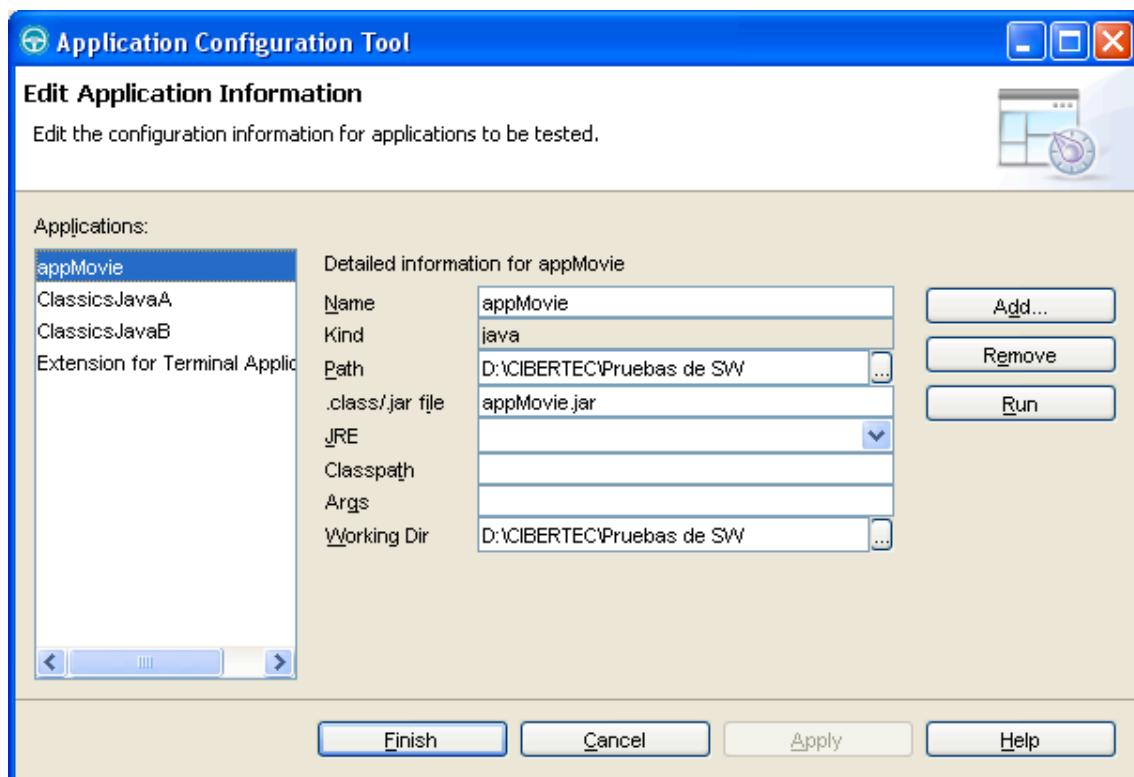


4. A continuación, la nueva aplicación aparecerá en la lista “Aplicaciones” de la ventana “Herramienta de Configuración de Aplicación” y su información detallada en el panel izquierdo (Nombre, Tipo, Vía de acceso, Archivo .class/.jar y Directorio de trabajo). Ahora, pulse **Run** para probar que la aplicación esté bien configurada. Luego, seleccione el botón **Apply**.

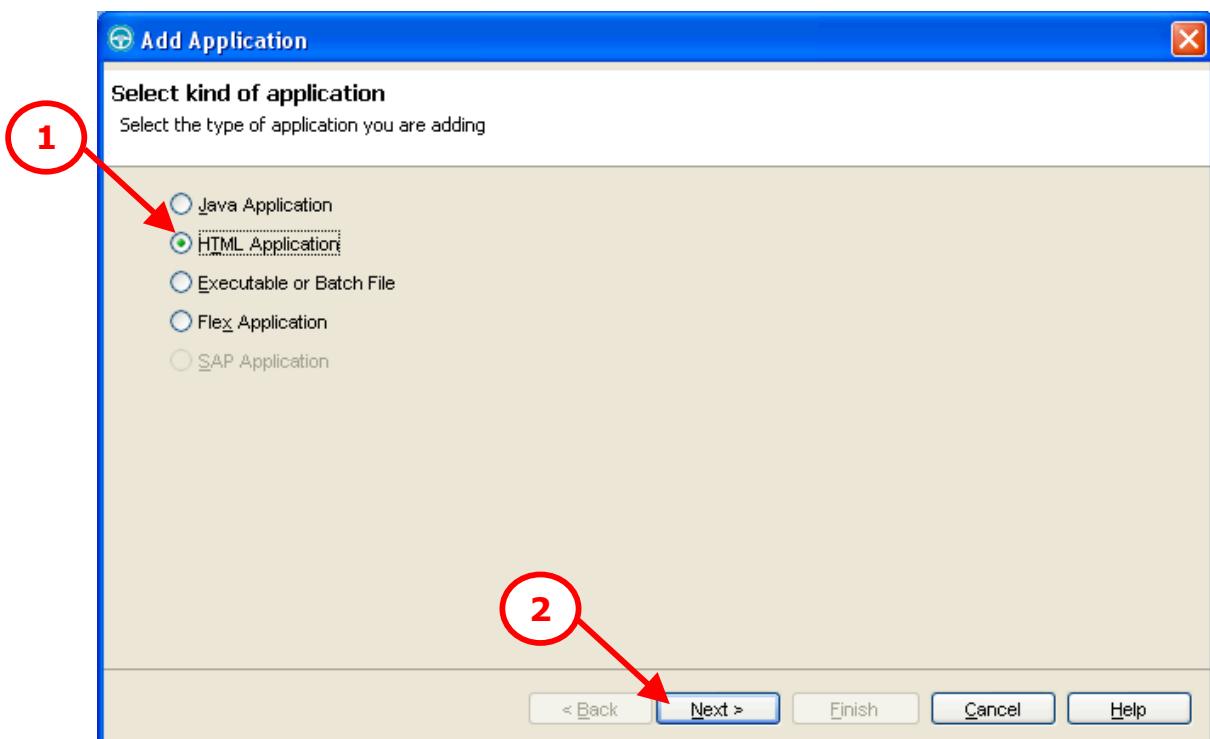


2.1.3.2. Agregar una aplicación HTML.

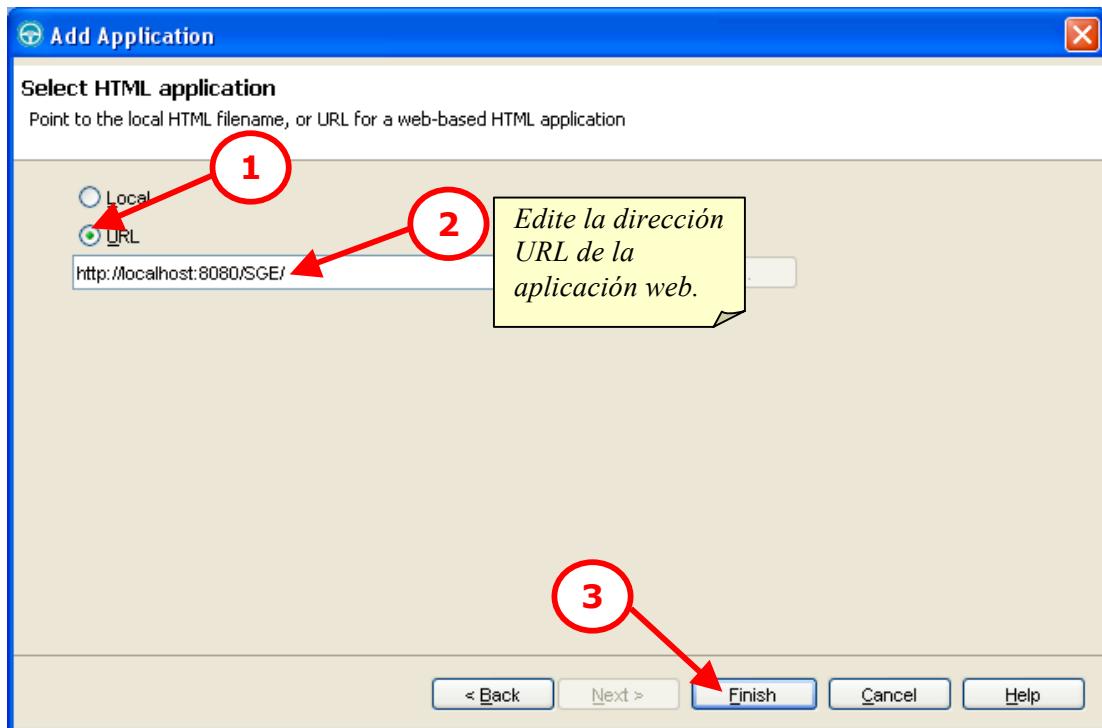
1. Desde la ventana “Herramienta de Configuración de Aplicación” pulse el botón **Add** para añadir una nueva aplicación HTML.



2. En el cuadro de diálogo “Añadir aplicación”, seleccione el tipo de aplicación. En este caso, seleccione **HTML Application** y luego pulse **Next**.

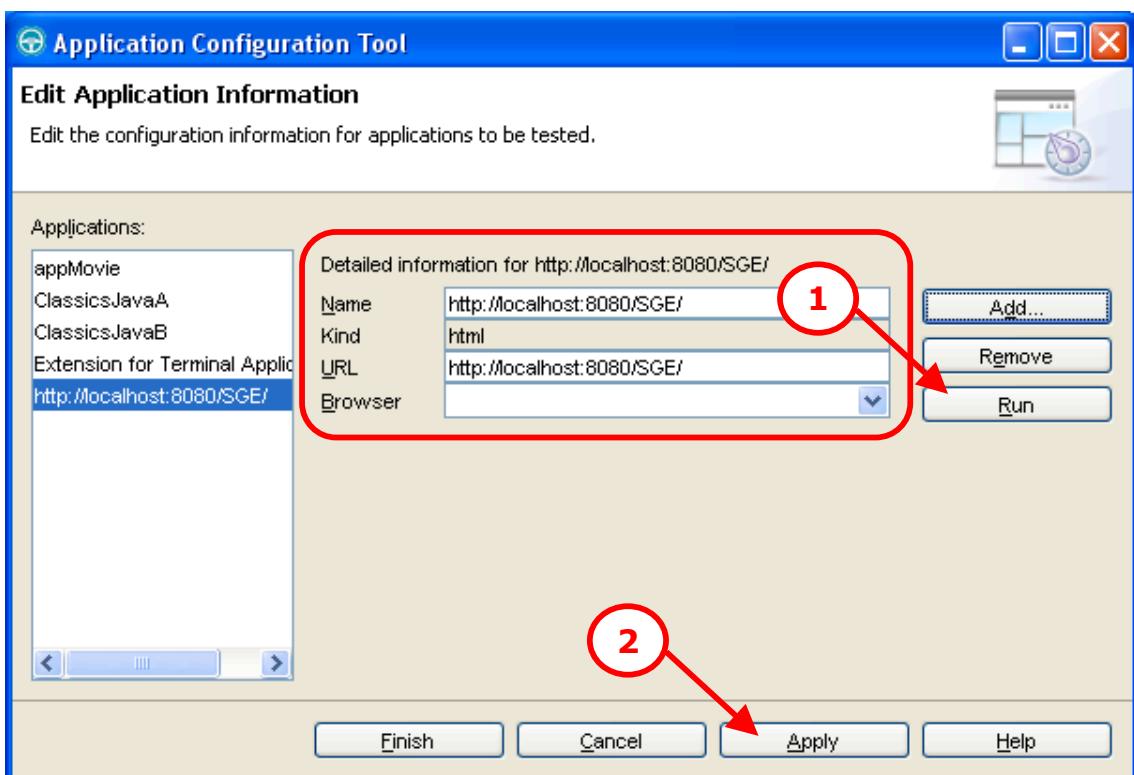


3. Ahora, especifique la página principal o el URL de la aplicación. Para ello, si elige **Local**, busque un archivo .htm o .html. Si elige **URL**, introduzca una dirección URL. Para el ejemplo, seleccione **URL**. Luego, pulse **Finish**.



4. La nueva aplicación aparecerá en la lista “Aplicaciones” de la ventana “Herramienta de Configuración de Aplicación” y su información detallada en el panel izquierdo (Nombre, Tipo y URL). Ahora, pulse **Run** para probar que la aplicación esté bien configurada. Luego, seleccione el botón **Apply**.

Nota: Asegúrese de Iniciar el contenedor web **Tomcat** que gestionará la ejecución de la aplicación.



2.1.4. Configuración de las propiedades del reconocimiento de objetos

A partir de la versión 8.0, el RFT introduce una nueva característica que permite configurar las propiedades de reconocimiento de objetos. Esta función utiliza la Herramienta de configuración de propiedades de objetos para configurar las propiedades de reconocimiento de objetos en la biblioteca de objetos personalizados. Durante la grabación de scripts de prueba, el archivo de biblioteca de objetos personalizados se utiliza como referencia para establecer las propiedades de reconocimiento de objetos y los pesos de las propiedades en el mapa de objetos, tal como se muestra en la figura 2.2.

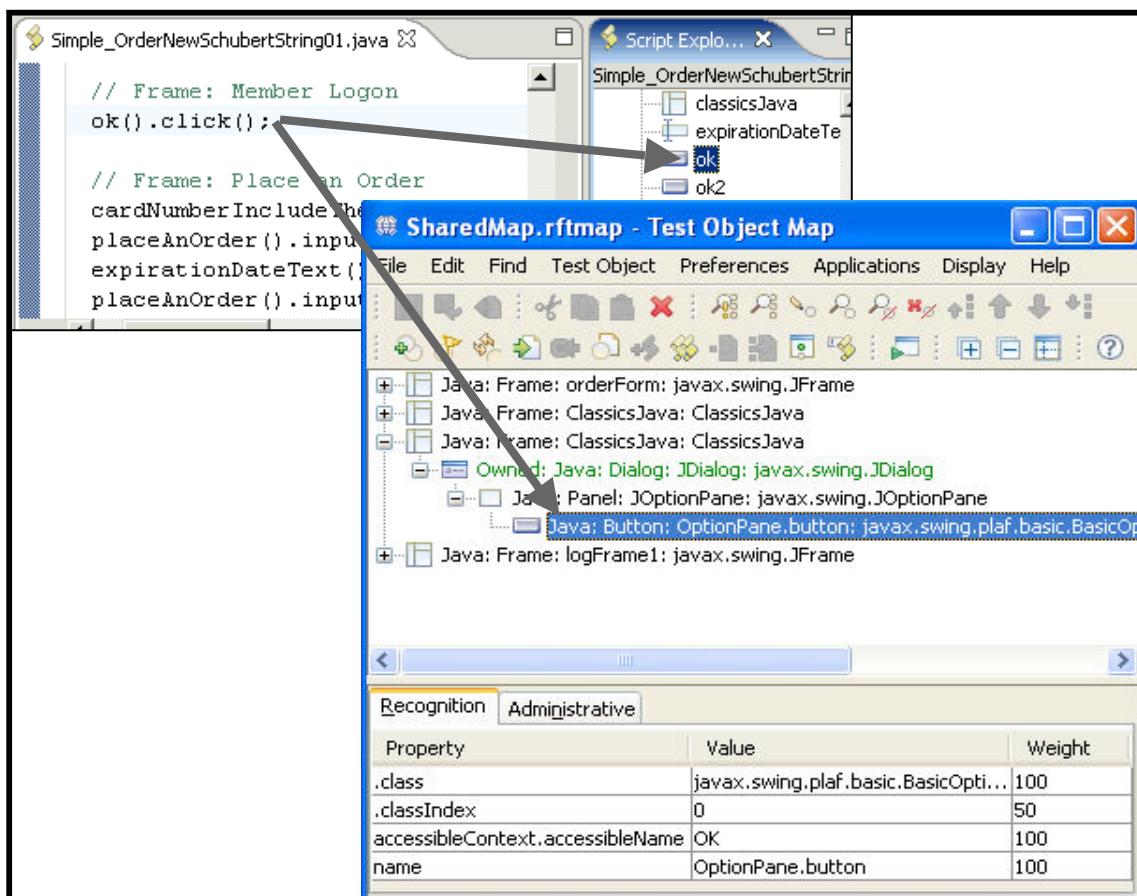


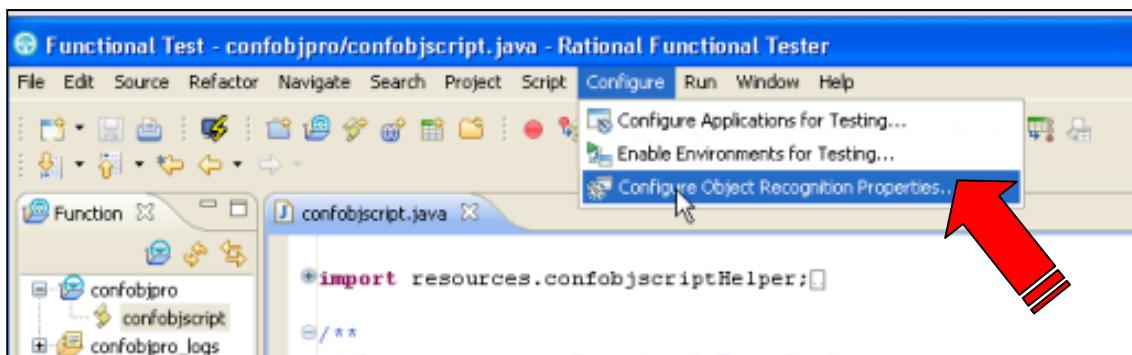
Figura 2.2. La información que describe los objetos de prueba es almacenada en un mapa de objetos de prueba

Un mapa de objetos de prueba es un catálogo jerárquico de descripciones de objetos de prueba. En la figura anterior se muestra dos pestañas que contienen las propiedades de un objeto de prueba: la pestaña “Reconocimiento” muestra los datos de reconocimiento que utiliza RFT para encontrar los objetos de prueba durante la reproducción, y la pestaña “Administrativo” muestra los datos administrativos internos del objeto.

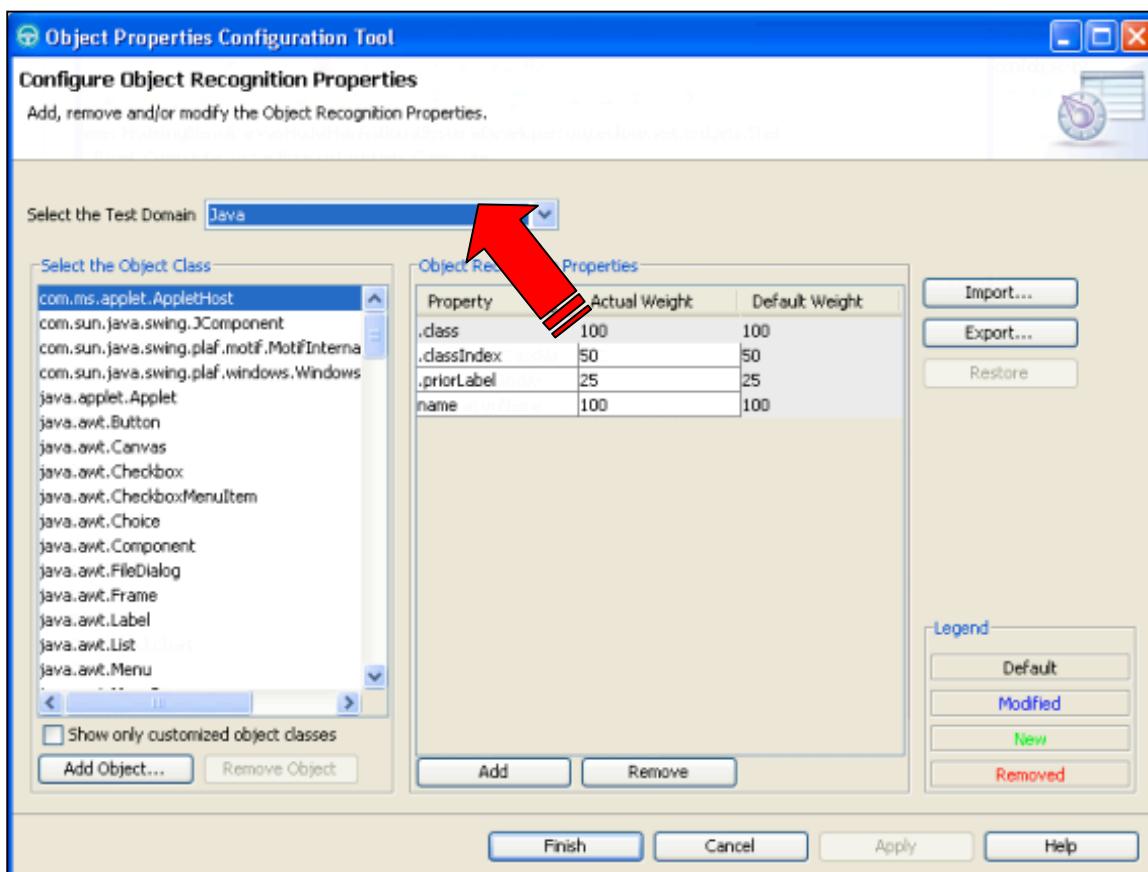
La Herramienta de configuración de propiedades de objetos por defecto muestra un catálogo de objetos para un dominio específico de prueba y objetos personalizados por el usuario que son utilizados por RFT. Personalizar las propiedades de reconocimiento de objeto ayuda a hacer que el script sea flexible a cambios en la aplicación de prueba.

En seguida, se muestra los pasos para ingresar a la herramienta de configuración de propiedades de reconocimiento de objetos que pertenecen al dominio de pruebas Java y se reflejará en todos los scripts a crear. El ejemplo muestra cómo eliminar el ancho y alto de un objeto.

1. Desde la barra de menú principal, seleccione **Configure > Configure Object Recognition Properties**.



2. En la siguiente ventana, seleccione Java como dominio de prueba.

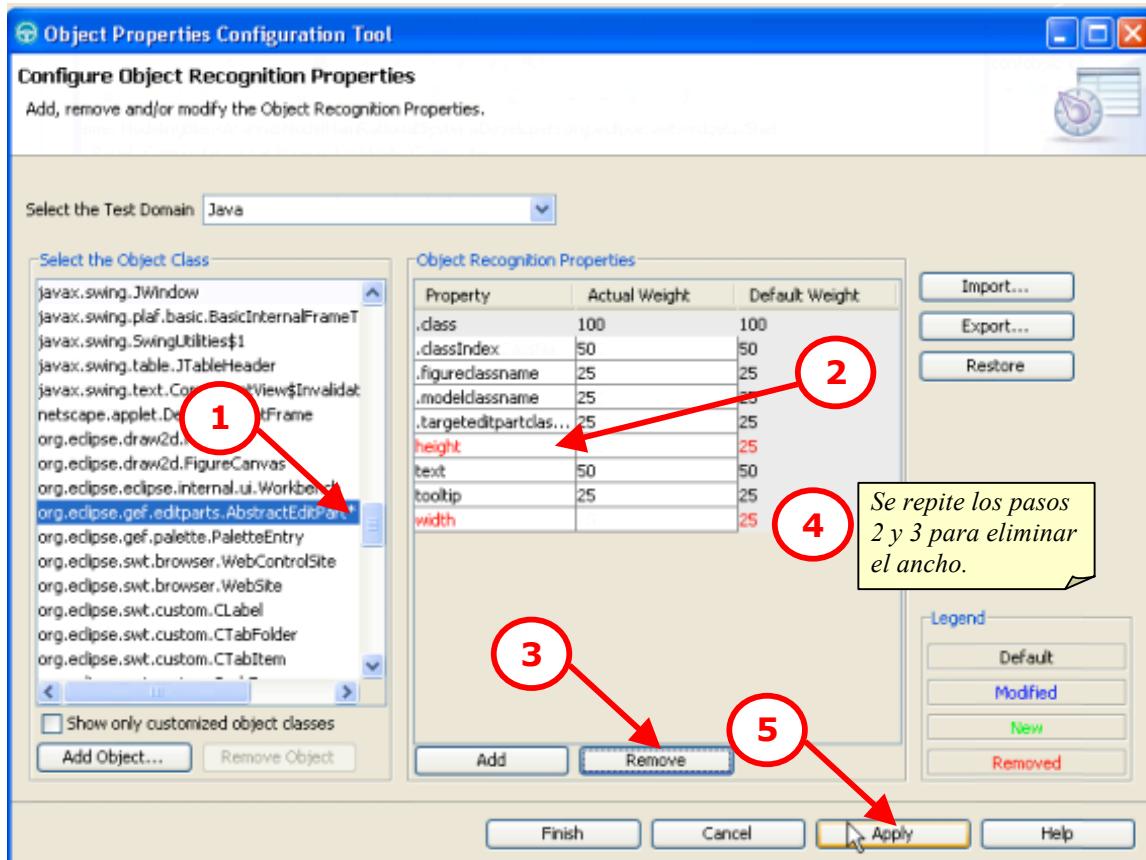


Los colores que se muestran en la leyenda hacen referencia al estado de las propiedades de un objeto:

- Default (color negro), es una propiedad por defecto
- Modified (color azul), si la propiedad se modificó
- New (color verde), si es una nueva propiedad
- Removed (color rojo), si la propiedad se eliminó

Nota: Si el objeto de prueba requerida de una aplicación bajo prueba no está en la lista, puede agregarlo a la biblioteca de objetos y personalizar sus propiedades de reconocimiento y pesos desde el botón **Add Object**.

3. A continuación, seleccione el objeto y las propiedades a eliminar. La eliminación de estas propiedades hacen que los scripts sean más resistentes a los cambios.



2.2. SCRIPT DE PRUEBAS FUNCIONALES

Un script de prueba en RFT puede ser simplificado cuando se crea a partir de un proceso de grabación, o de Java si se crean directamente con código Java. El primer tipo de script genera un archivo que contiene las acciones del usuario en la aplicación bajo prueba y otro archivo con el script en código Java. La siguiente figura muestra los pasos de alto nivel para grabar, reproducir y visualizar los resultados de un script simplificado de pruebas funcionales en RFT:

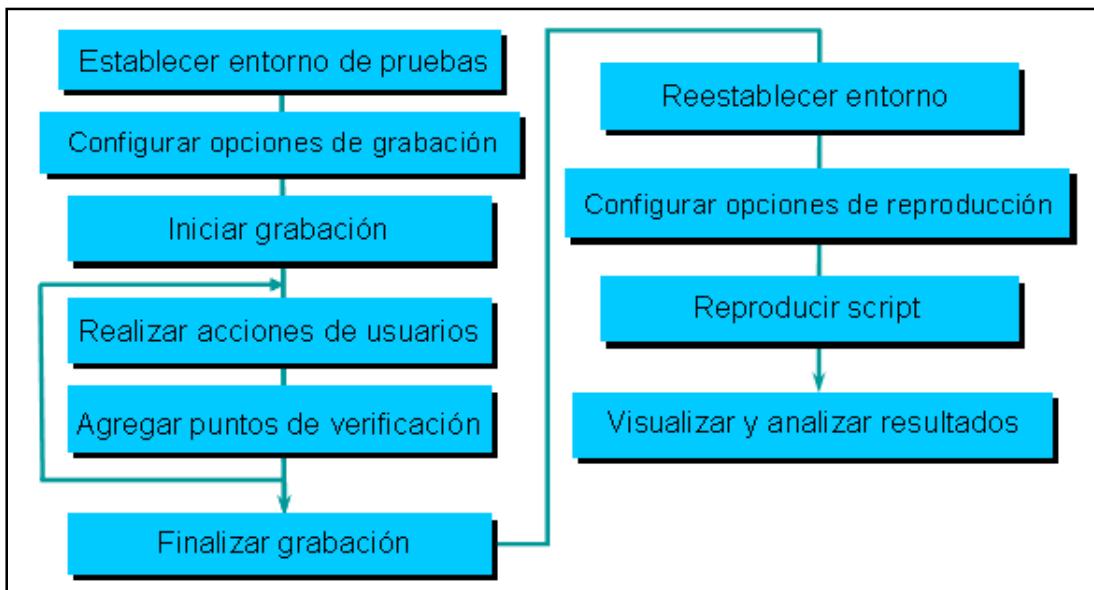


Figura 2.3 Proceso de grabación, reproducción y visualización de resultados de un script de pruebas funcionales

Como se explicó en el apartado anterior, el proceso de grabación siempre genera un script, el cual se asocia con un mapa de objetos de prueba.

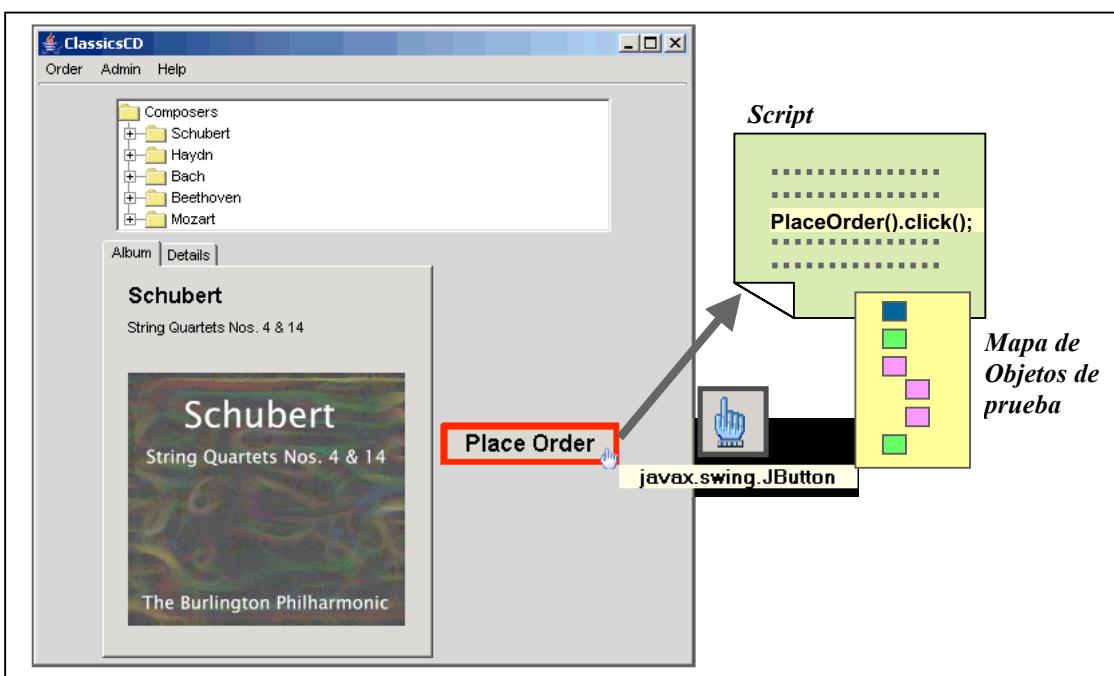


Figura 2.4. El proceso de grabación produce un script y un mapa de objetos de prueba

Un mapa de objetos de prueba puede ser privado o compartido.

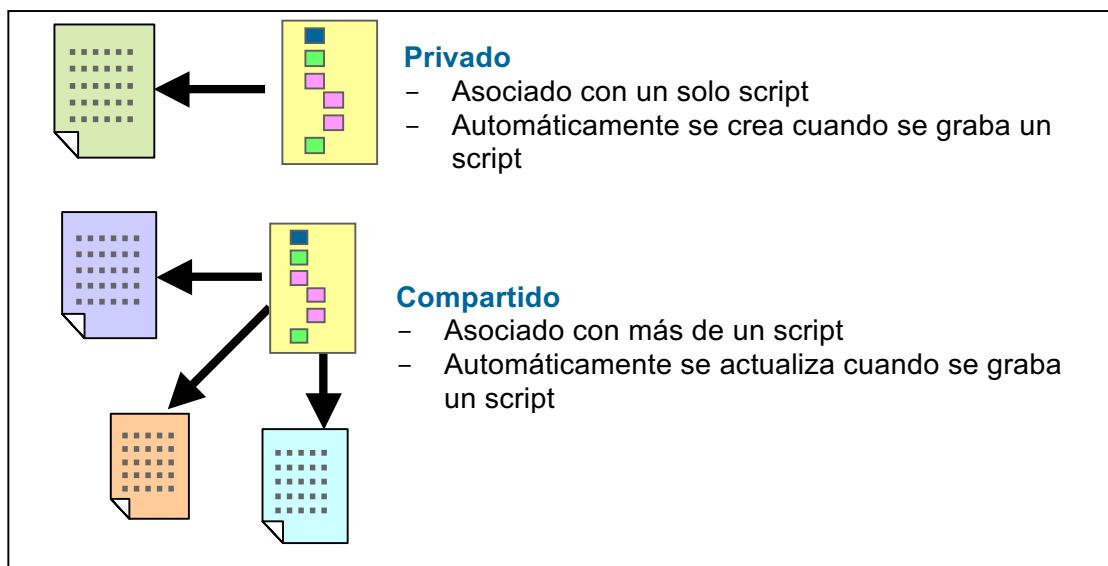


Figura 2.5. Mapa de objetos de prueba

En los siguientes puntos, se detallará el proceso de grabación, el proceso de reproducción y visualización de resultados.

2.2.1 Grabación de un script

Antes de iniciar el proceso de grabación, asegúrese de haber configurado el entorno de pruebas y la aplicación a probar.

Durante el proceso de grabación, se puede agregar **comandos controlados por datos** a fin de crear un pool de datos para probar otros valores durante el proceso de reproducción, y **puntos de verificación** para probar el estado de un objeto de la GUI.

La siguiente figura muestra el monitor con una barra de herramientas que se activa durante el proceso de grabación:

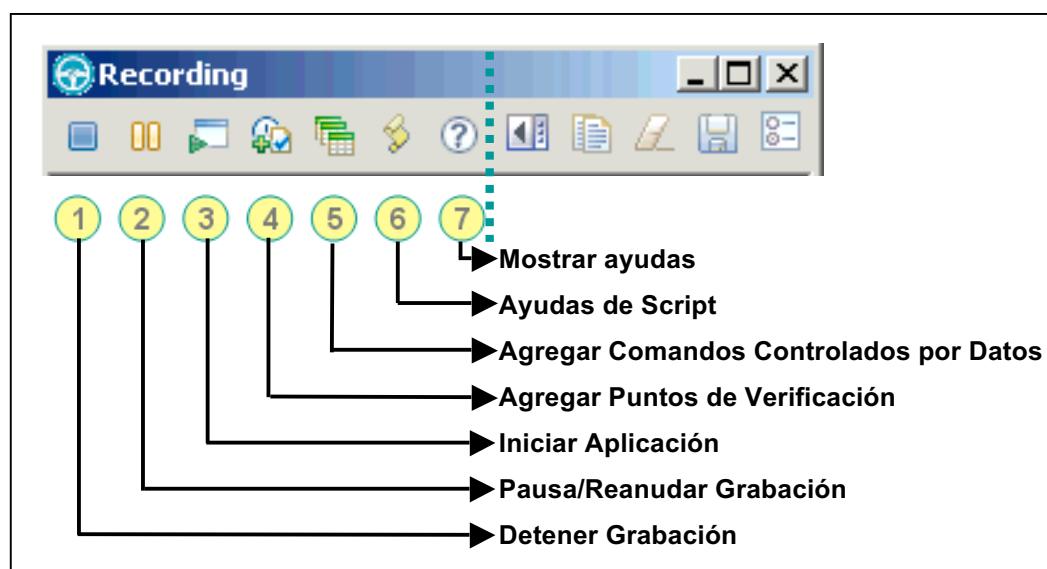


Figura 2.6. Barra de herramientas del monitor de grabación

2.2.1.1 Comandos controlados por datos.

Este elemento creará un pool de datos con las variables que hacen referencia a un conjunto de objetos que contienen datos. Con RFT puede agregar comandos controlados por datos durante el proceso de grabación o desde la vista de aplicación una vez se haya generado el script.

2.2.1.2 Puntos de verificación.

Un punto de verificación (PV) es la respuesta que esperamos del sistema. Esta respuesta muestra algún resultado sobre un objeto de la GUI, ejemplos:

- Sumas, totales, balances sobre una etiqueta o campos de texto
- Datos cargados sobre campos de texto de un formulario
- Mensajes de error sobre una etiqueta de un cuadro de diálogo
- Resultados de consultas sobre una tabla
- Imágenes

En RFT, un PV captura la información del objeto y los valores literales de la aplicación de prueba y la almacena, en la línea base, a efectos de comparación durante la reproducción. Cuando se reproduce un script, un punto de verificación vuelve a capturar la información del objeto para compararla con la línea base y para ver si se ha efectuado algún cambio, ya sea o no de forma intencionada. Comparar la información del objeto real de un script con la línea base resulta útil para identificar posibles defectos.

En RFT, se graba un PV mediante la selección de una de las 5 acciones siguientes sobre el objeto que deseamos evaluar:

- Para verificar datos.
- Para verificar propiedades del objeto
- Obtener el valor de una propiedad específica del objeto
- Esperar a seleccionar el objeto a probar. Esta opción se utiliza en caso sea necesario especificar un tiempo de espera para visualizar el resultado que se cargará sobre un objeto. Esto es para evitar problemas en la visualización de resultados.
- Visualizar cambios sobre una imagen. Al grabar este PV, se crea un archivo de imagen de línea base. Cada vez que reproduce el script, la imagen se comparará para ver si se han producido cambios, ya sea de forma intencionada o no intencionada. **La verificación de la imagen se hace estrictamente píxel a píxel.** Esto resulta útil para identificar errores posibles.

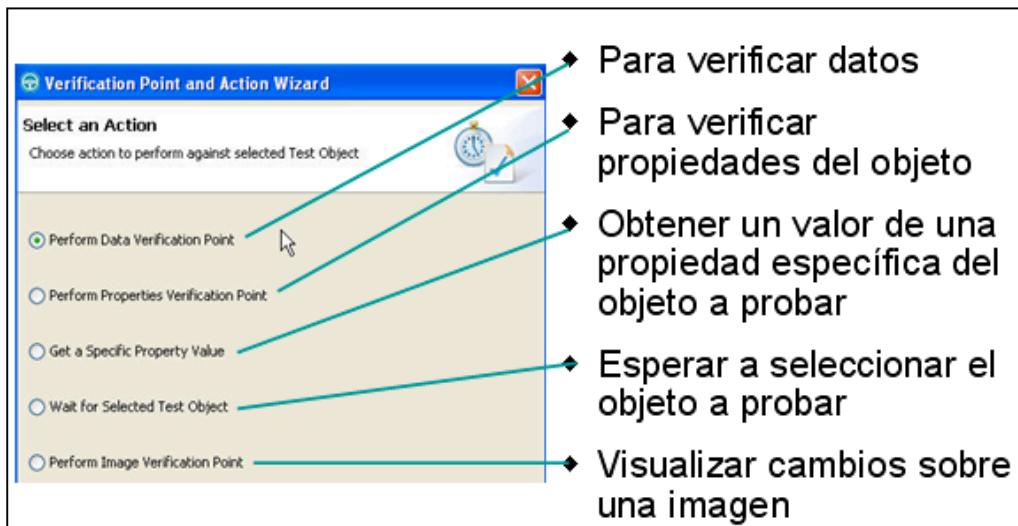


Figura 2.7. Tipos de Puntos de Verificación

A continuación, se muestran algunos ejemplos de PV:

Item:	Bach	Sub-Total: \$14.99
	Violin Concertos	Related Items: \$0.00
Quantity:	4	S&H: \$1.00
		Total: \$60.96
\$60.96 (JLabel)		

Figura 2.8. PV para verificar datos de una etiqueta



Figura 2.9. PV para verificar propiedades de una etiqueta

AÑO ACADÉMICO:	2010
MOTIVO:	SELECCIONE UN MOTIVO
Debe adjuntar un documento que sustente la solicitud:	
<input type="file"/> Examinar...	

Figura 2.10. PV para verificar propiedades de un file text (Componente HTML)

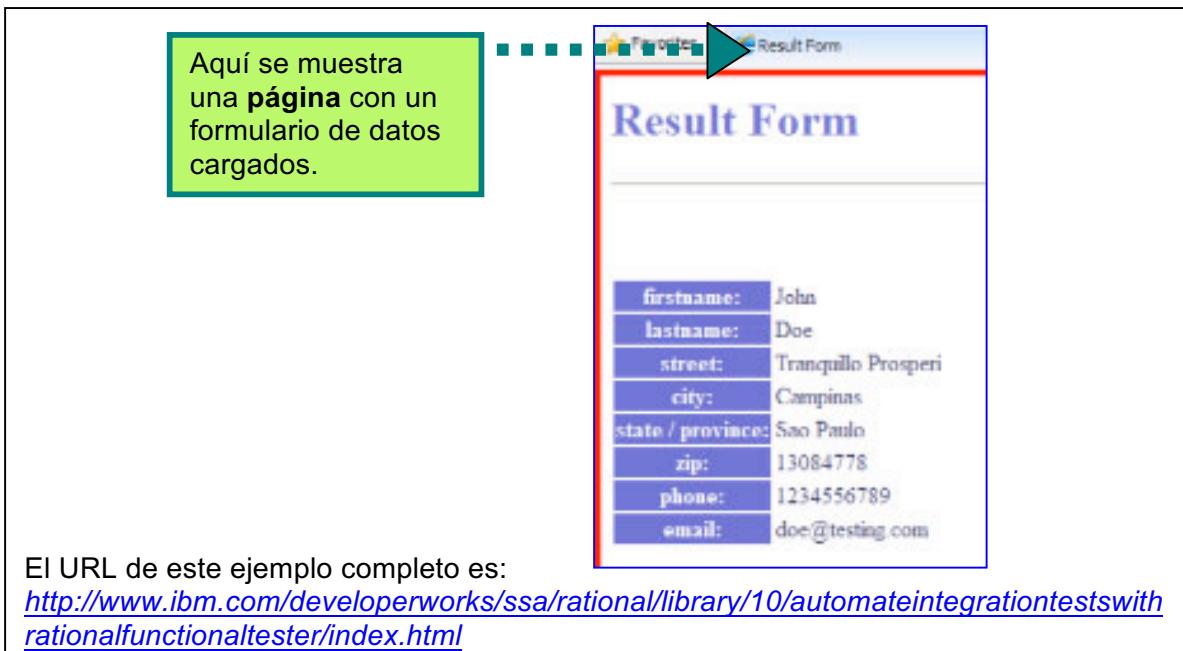


Figura 2.11. PV para verificar el título de una página (valor de una propiedad específica) que tiene el formulario con datos cargados

2.2.1.3 Pasos para grabar un script.

Los pasos para grabar un script son los siguientes:

- Inicio de grabación** {
- 1.Crear un proyecto
 - 2.Crear el script
 - 3.Seleccionar la aplicación bajo prueba
 - 4.Realizar acciones de usuarios
 - 5.Añadir comandos controlados por datos
 - 6.Agregar puntos de verificación
 - 7.Finalizar proceso de grabación

Nota: Es normal, por primera vez, que los usuarios realicen acciones incorrectas durante el proceso de grabación. Cualquier error del usuario podrá ser corregido después, así es que no será necesario detener la grabación si se comete algún error.

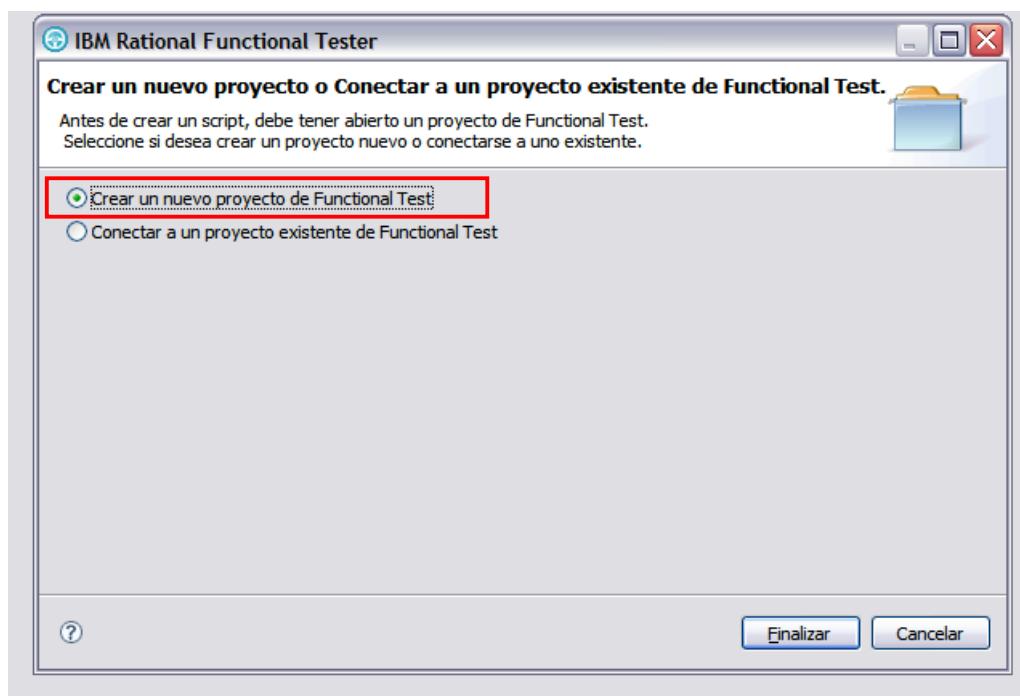
De inmediato, realizará los pasos citados grabará para grabar un script de prueba simplificado para probar una aplicación de escritorio Java.

Inicio de grabación

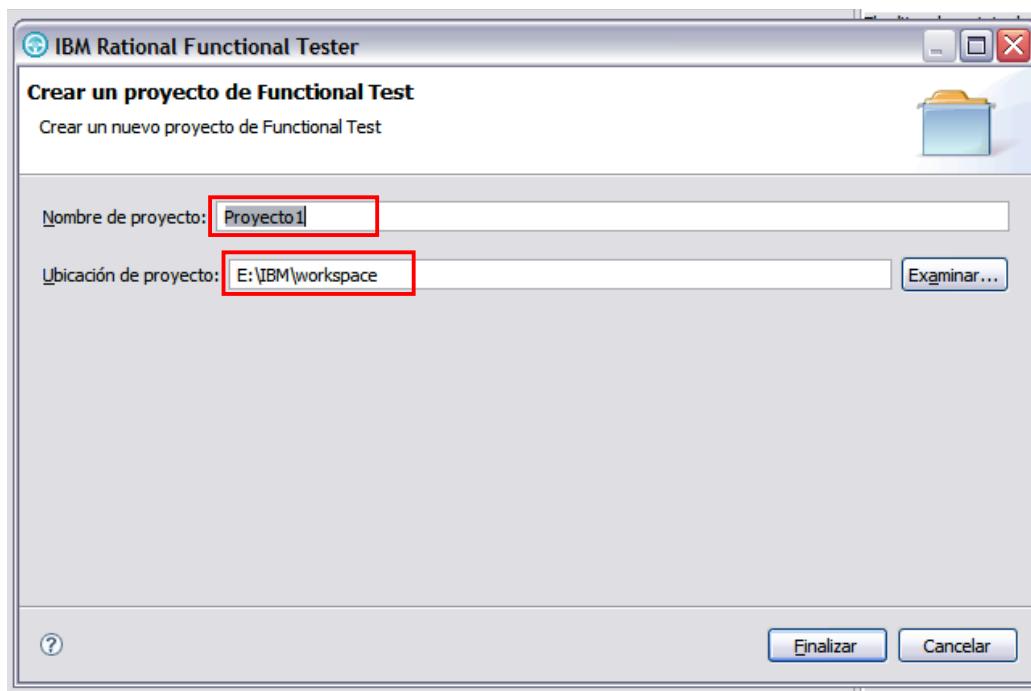
1. Para comenzar el proceso de grabación, pulse el icono Grabar un script de *Functional Test* (●) desde la barra de herramientas.



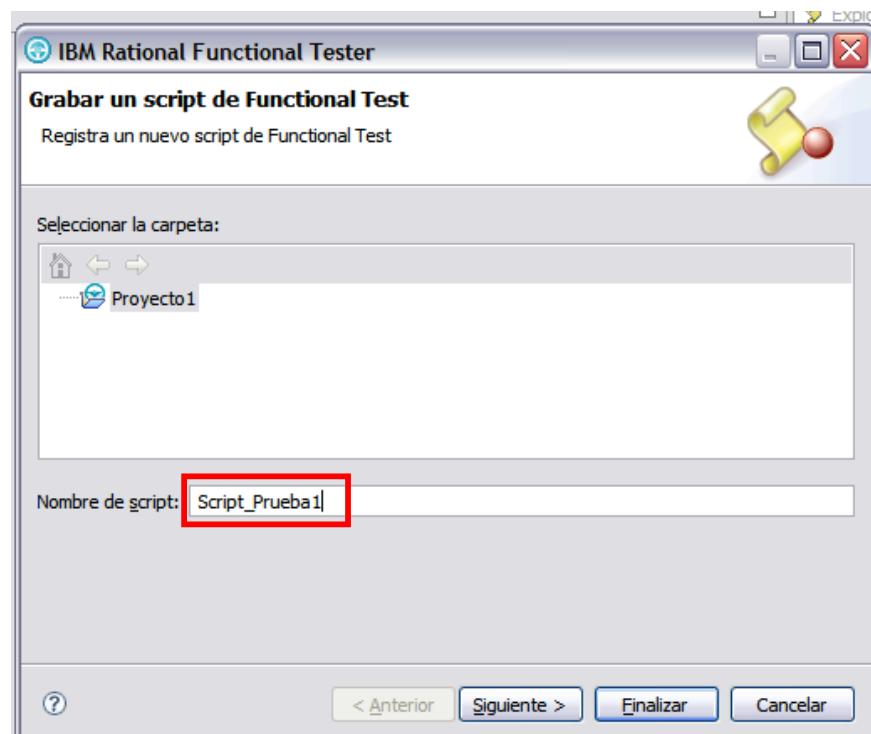
2. Al hacer click a “Grabar un Script” deberá seleccionar el proyecto a grabar. Seleccione “Crear un nuevo proyecto de *Functional Test*” y pulse Finalizar.



3. Edite el nombre del Proyecto, la ubicación de Proyecto y pulse Finalizar.



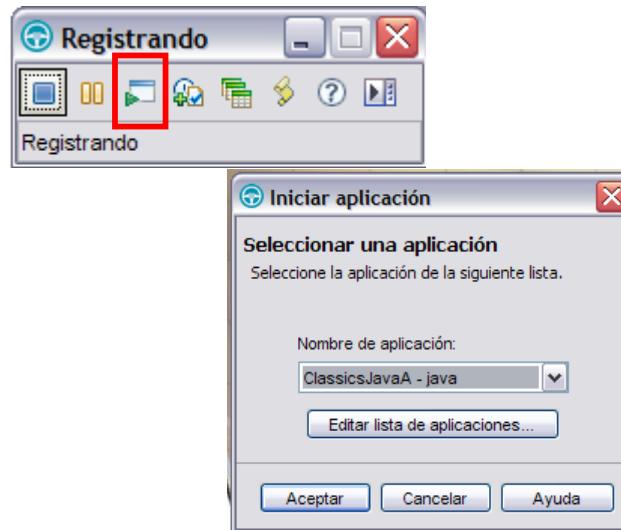
4. En la ventana “Grabar un script de *Functional Test*”, coloque el nombre del Script y pulse Finalizar.



5. La ventana de Functional Tester se minimiza automáticamente, y se muestra el Monitor de grabación.



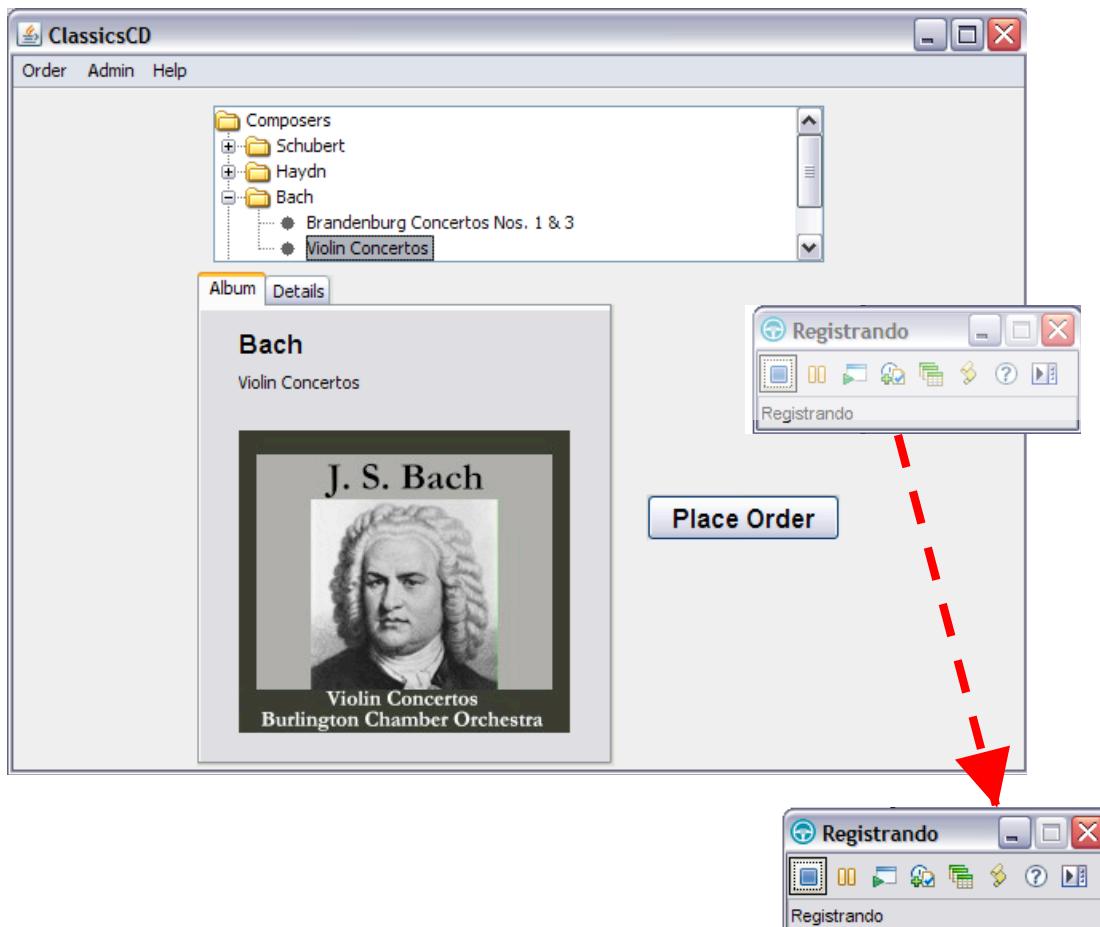
6. Para iniciar la aplicación de prueba, pulse el ícono Iniciar aplicación ().



7. En la ventana “Iniciar aplicación”, seleccione **ClassicsJavaA - Java** y a continuación pulse Aceptar.



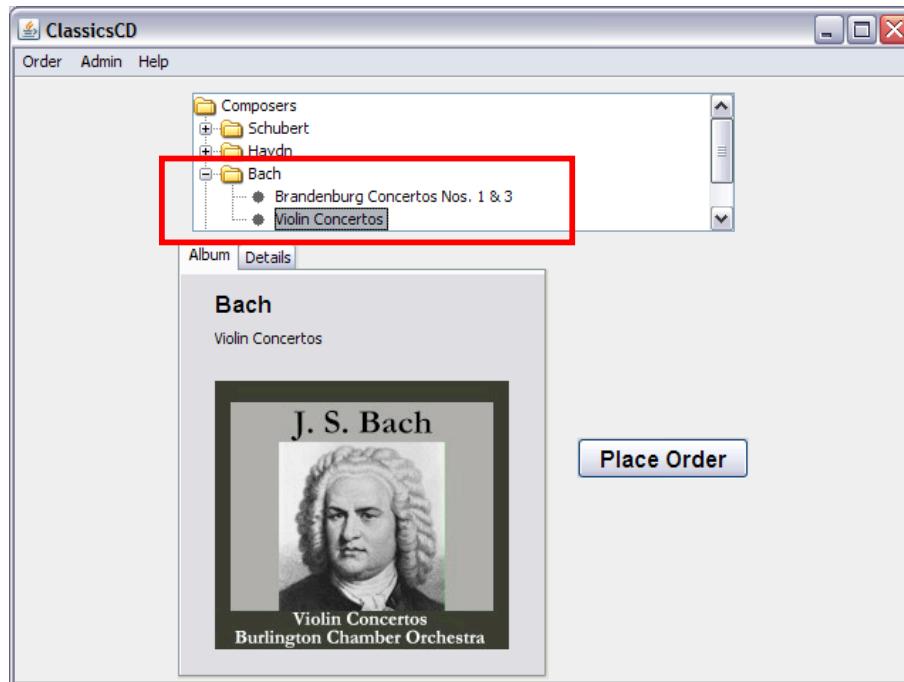
8. Se abre la aplicación de ejemplo, ClassicsCD. Si el monitor de grabación aparece delante de la aplicación, puede arrastrarlo hasta el ángulo inferior derecho de la pantalla.



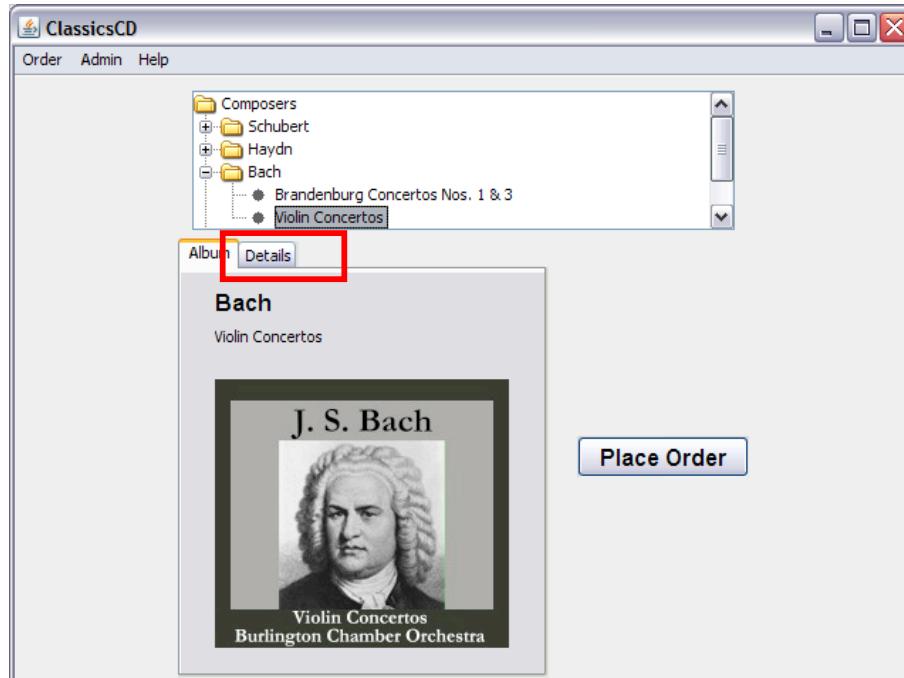
Grabación de acciones

Ahora realizará las acciones de usuario sobre la aplicación **ClassicsJavaA** para generar un pedido.

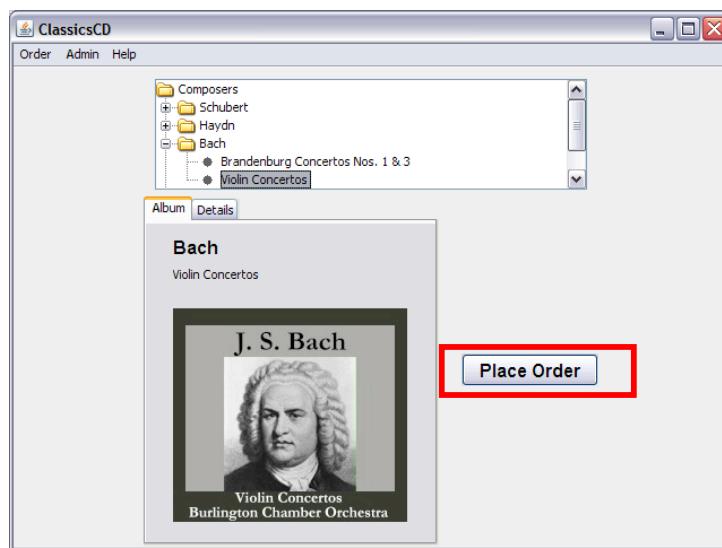
- Pulse el signo (+) situado al lado de Bach para abrir la lista de CD que estén en venta de dicho compositor y, a continuación, pulse Violin Concertos.



- Ahora, pulse la página “Detalles” para ver la descripción del álbum.



3. Pulse Hacer pedido.



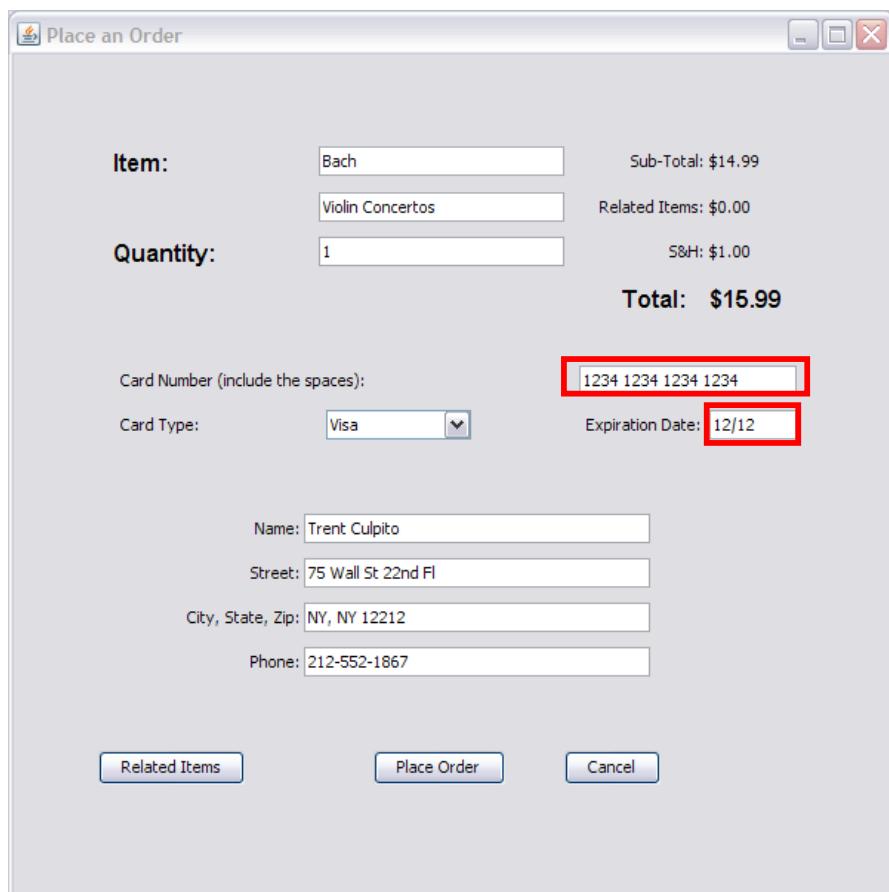
4. En la ventana “Member Logon”, conserve los valores predeterminados del Cliente existente Trent Culpito



5. Escriba xxxx en el campo Contraseña y pulse OK



6. En la ventana “Hacer Pedido”, escriba 1234 1234 1234 1234 en el campo número de tarjeta y, a continuación, escriba 12/12 en el campo fecha de expiración.



Comandos controlados por datos

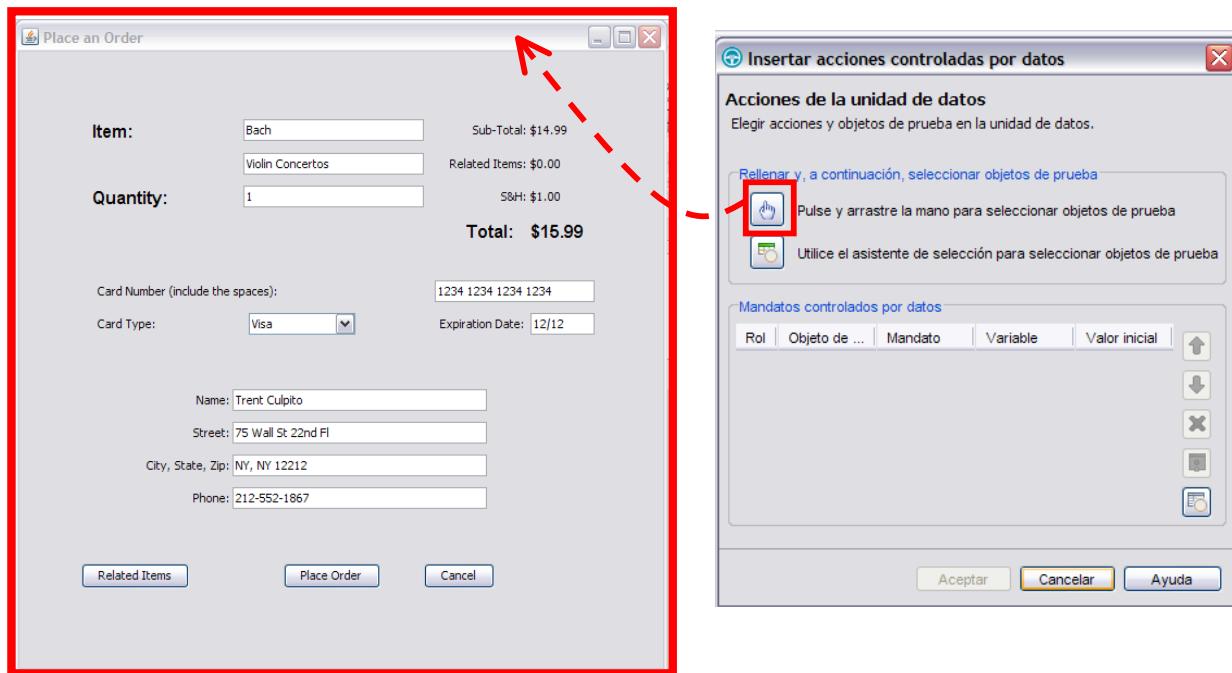
Ahora, agregue comandos controladas por datos a fin de crear un pool de datos con los datos procedentes de los objetos de la GUI.

Nota: Un pool de datos es una recopilación de registros de datos relacionados. Proporciona valores de datos a las variables de un script de prueba durante la reproducción de éste.

1. En la barra de herramientas del monitor de grabación, pulse Insertar comandos controlados por datos ().

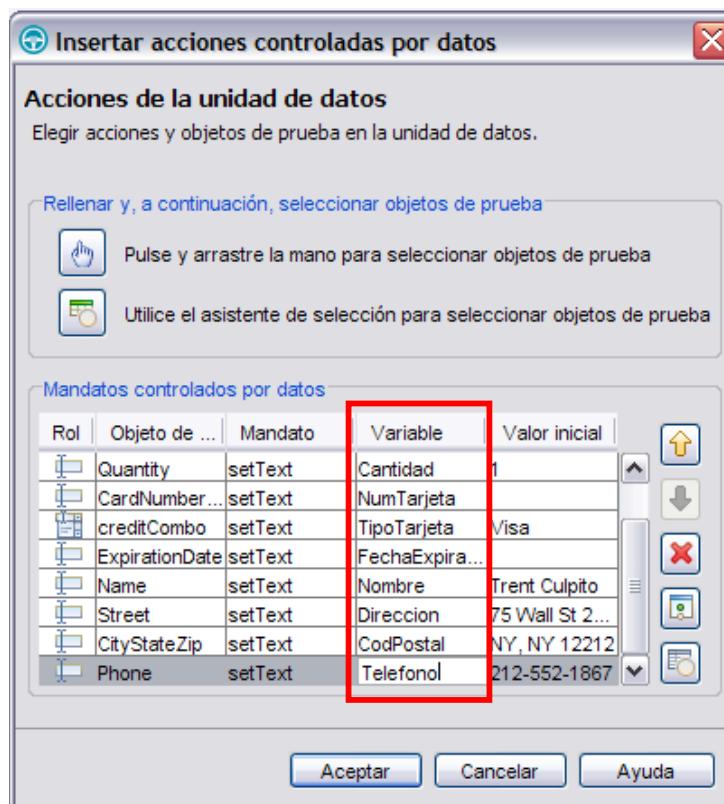


2. En la página Insertar “Acciones controladas por datos”, arrastre el buscador de objetos () a la barra de título de la ventana “Hacer Pedido” de la aplicación. Aparecerá un contorno de color rojo alrededor de la ventana “Hacer Pedido”.

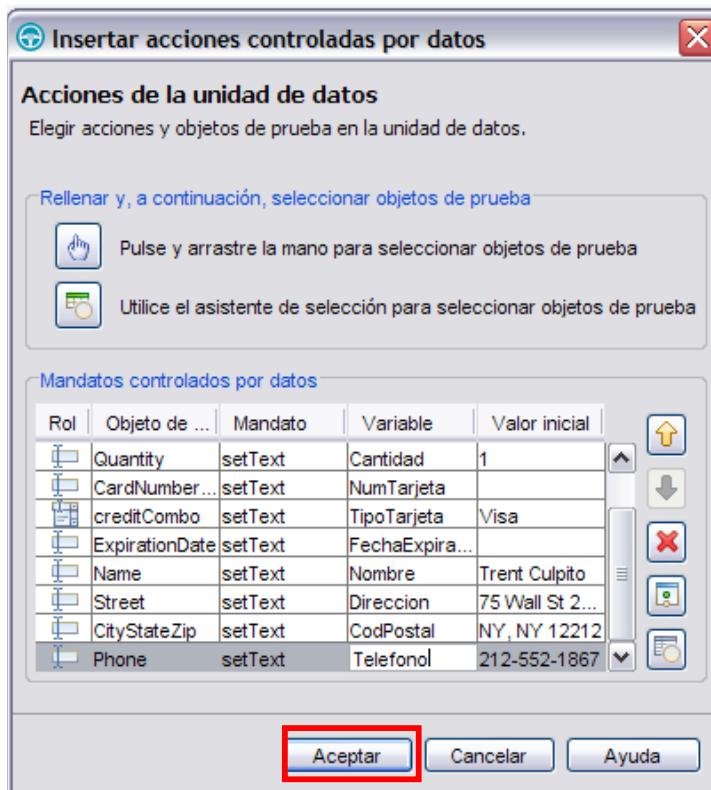


3. Suelte el botón del ratón. En la ventana “Acciones controladas por datos”, se mostrará la información sobre los controles de la GUI seleccionados. Reemplace los valores de la columna Variable por otros textos en español, pues esta columna será la cabecera del pool de datos que se creará. Utilice los siguientes nombres para la lista de variables; no utilice espacios en los nombres:

- Compositor
- Artículo
- Cantidad
- NumTarjeta
- TipoTarjeta
- FechaExpiracion
- Nombre
- Direccion
- CodPostal
- Telefono



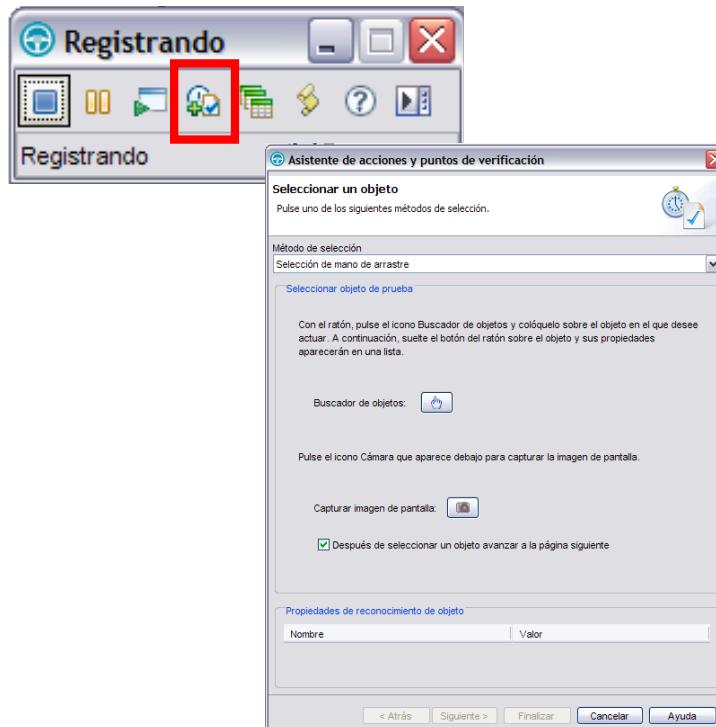
4. Ahora, pulse Aceptar.



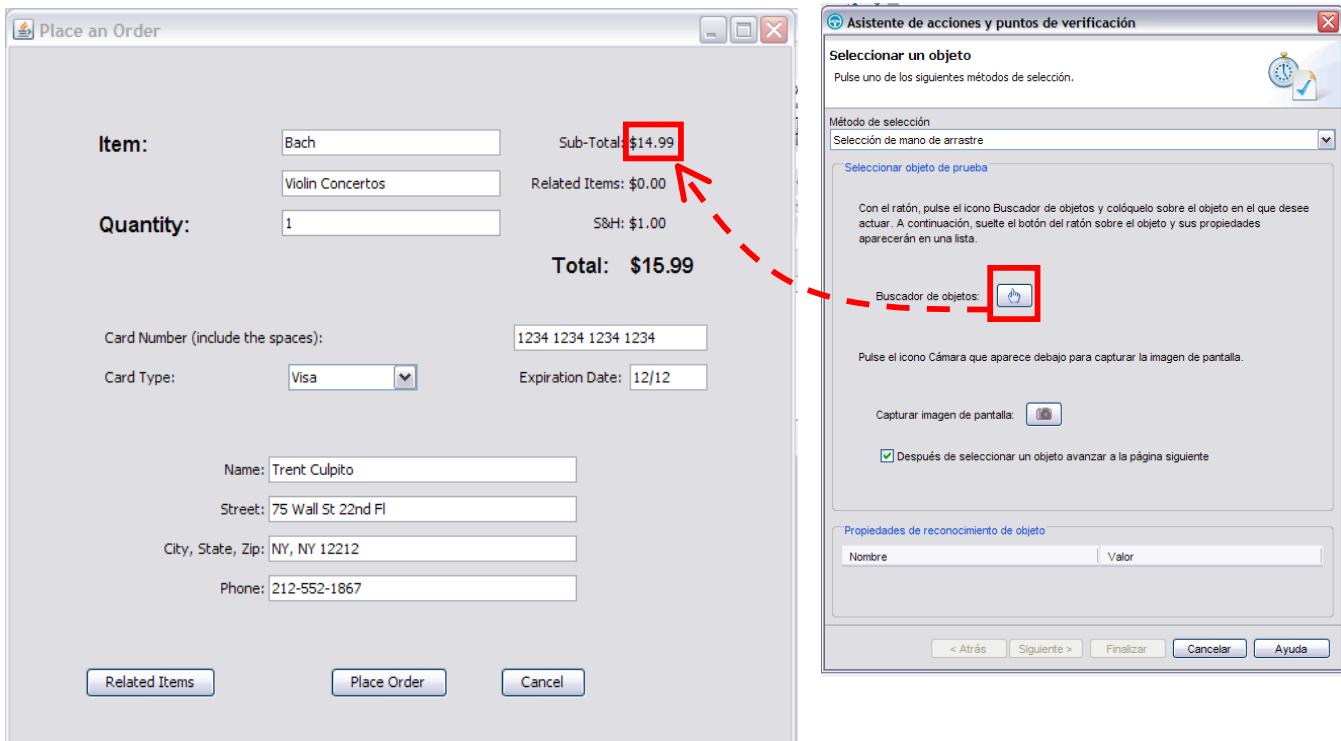
Puntos de verificación

En esta sección, creará un punto de verificación con una referencia de agrupación de datos para comprobar que el importe del CD a comprar es correcto.

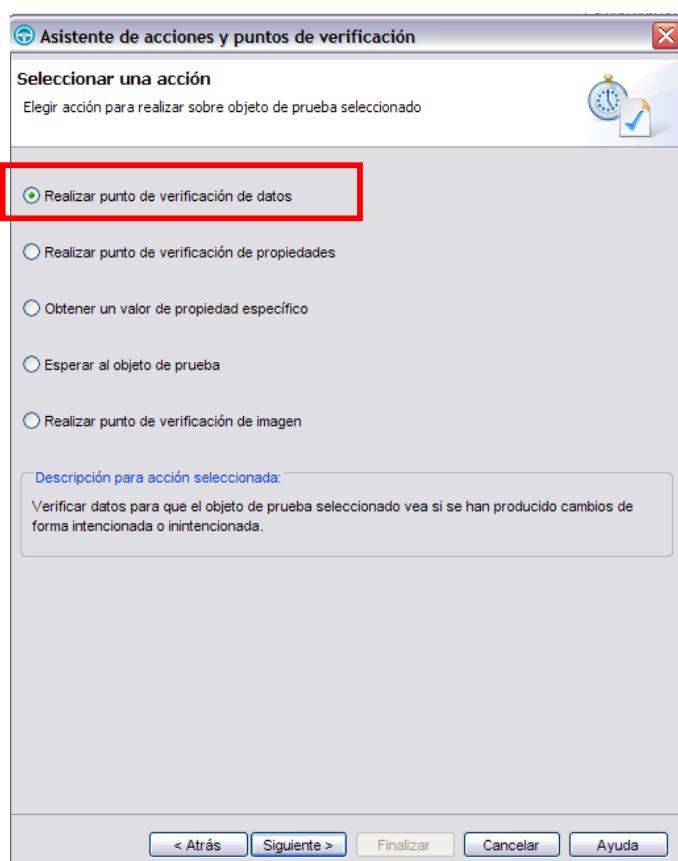
1. Desde el monitor de grabación, seleccione Insertar punto de verificación y mandato de acción (); se mostrará un asistente.



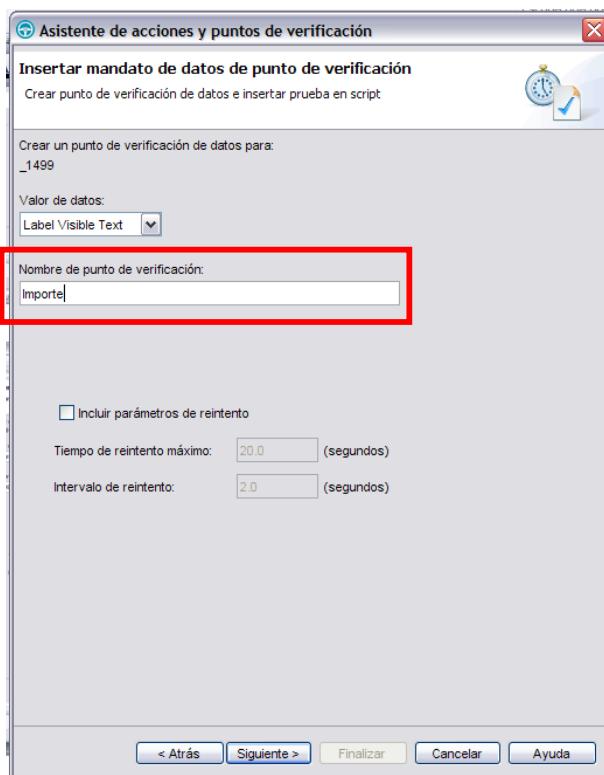
2. En el “Asistente de Acciones y puntos de verificación”, arrastre el buscador de objetos () hasta el valor \$14.99, que aparece junto a Sub-Total en la aplicación ClassicsCD. Aparece un contorno de color rojo en la cantidad, \$14.99.



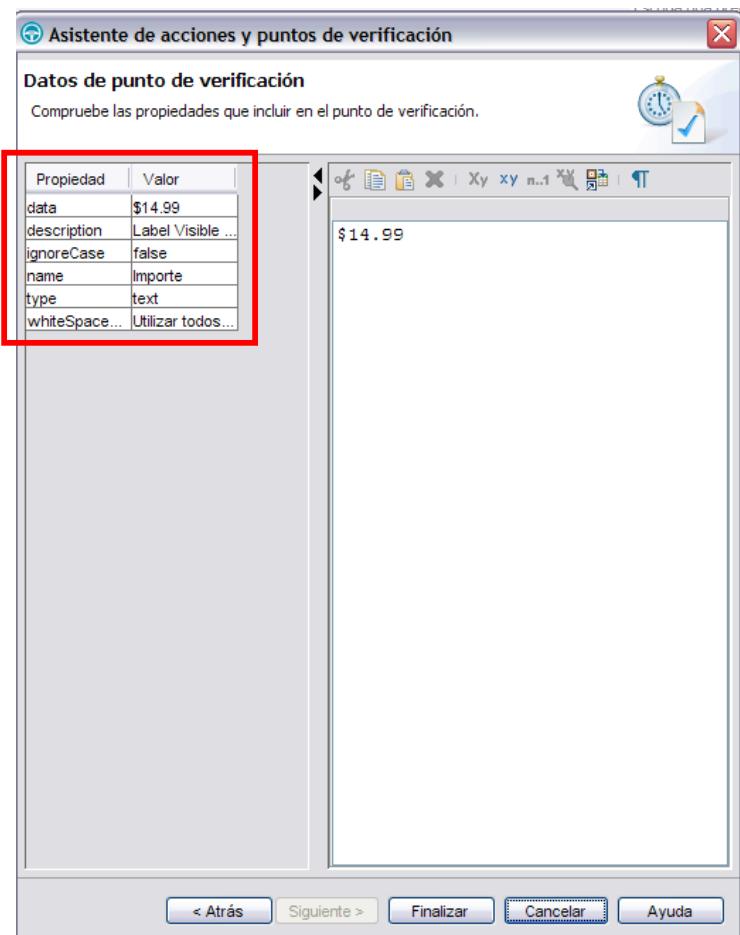
3. A continuación, pulse “Realizar punto de verificación de datos” para probar si el importe del CD cambia según la cantidad y pulse Siguiente.



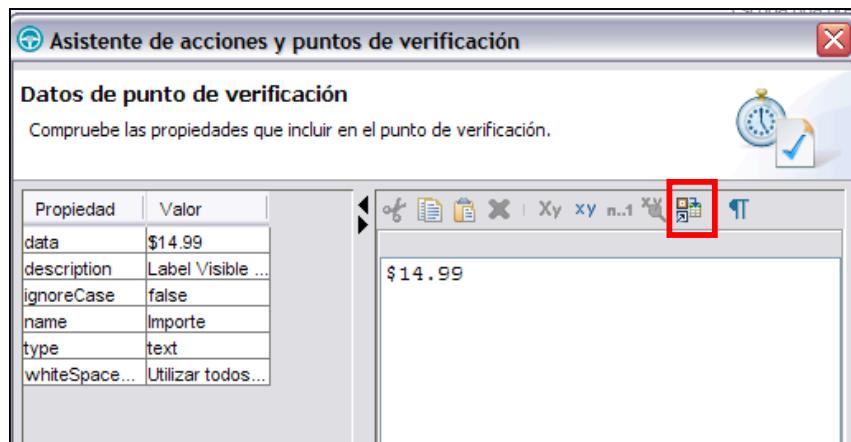
4. Ahora, escriba Importe en Nombre de puntos de verificación y pulse Siguiente.



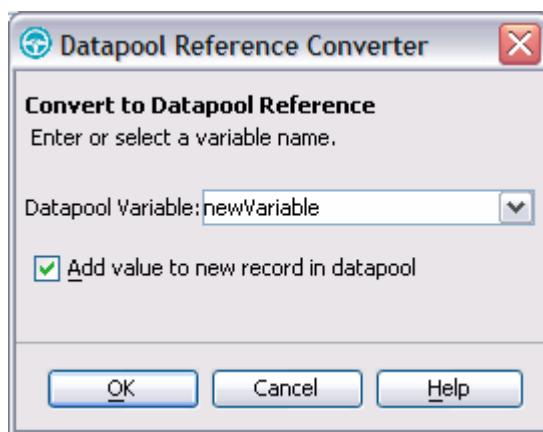
5. A continuación, se muestra la información del objeto que se le aplicó el PV. Luego, pulse Finalizar.



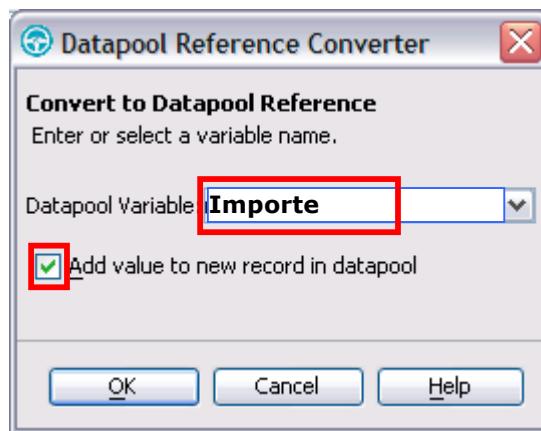
6. En la barra de herramientas de la página “Datos de punto de verificación”, pulse Convertir valor en referencia de agrupación de datos (grid icon) para utilizar un pool de datos en lugar de un valor literal en un punto de verificación. (Si no puede ver la opción Convertir valor en referencia de agrupación de datos en la barra de herramientas, aumente el tamaño de la página arrastrándolo hacia la derecha).



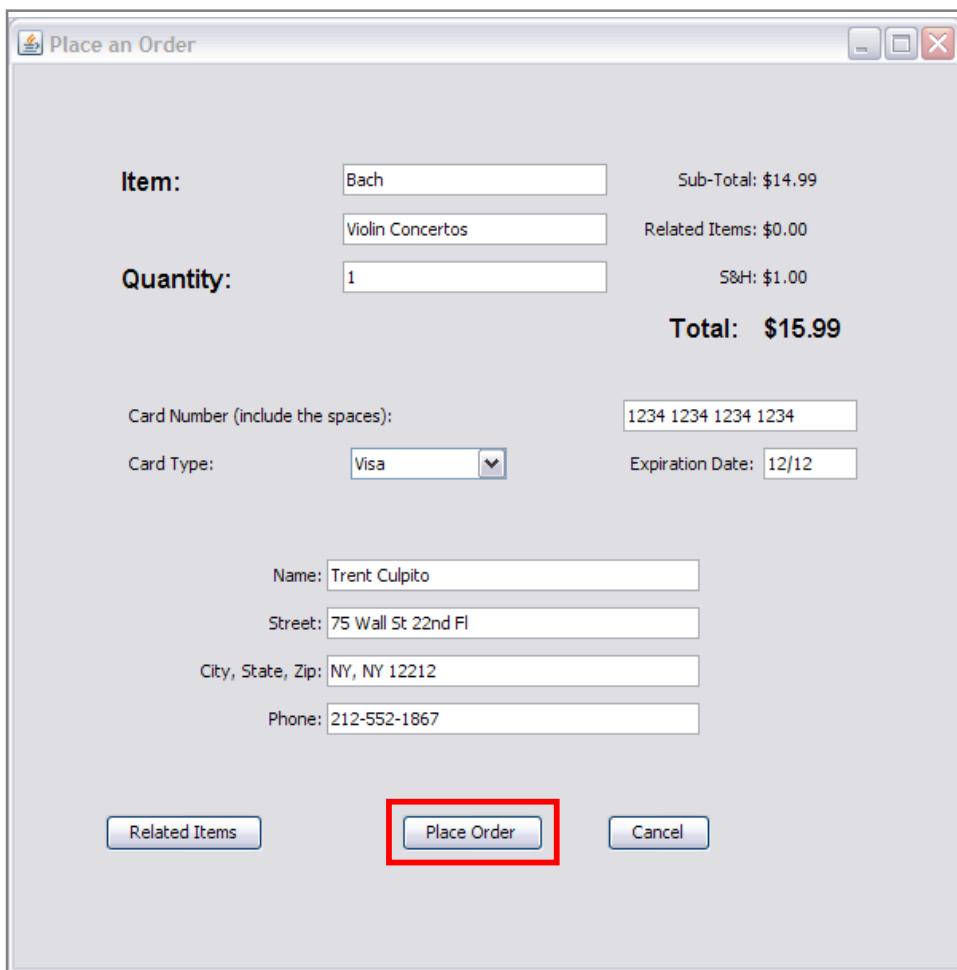
7. Se abre el cuadro de diálogo Convertidor de referencias de agrupación de datos.



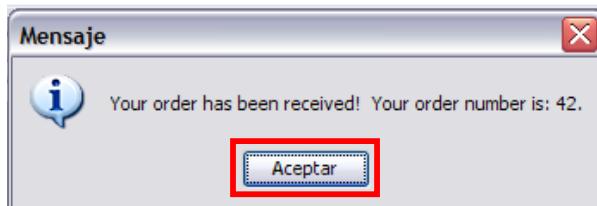
8. En Variable de agrupación de datos, escriba **Importe** para sustituir el valor **newVariable** como cabecera del pool de datos. Luego, asegúrese de seleccionar **Añadir valor al nuevo registro de la agrupación de datos** para añadir la variable Importe al registro del pool de datos existente que se ha creado en el apartado anterior. Por último, pulse OK.



9. Para hacer un pedido, en la aplicación ClassicsCD, pulse Hacer pedido.



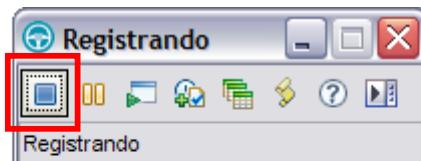
10. A continuación, pulse Aceptar para cerrar el mensaje que confirma el pedido.



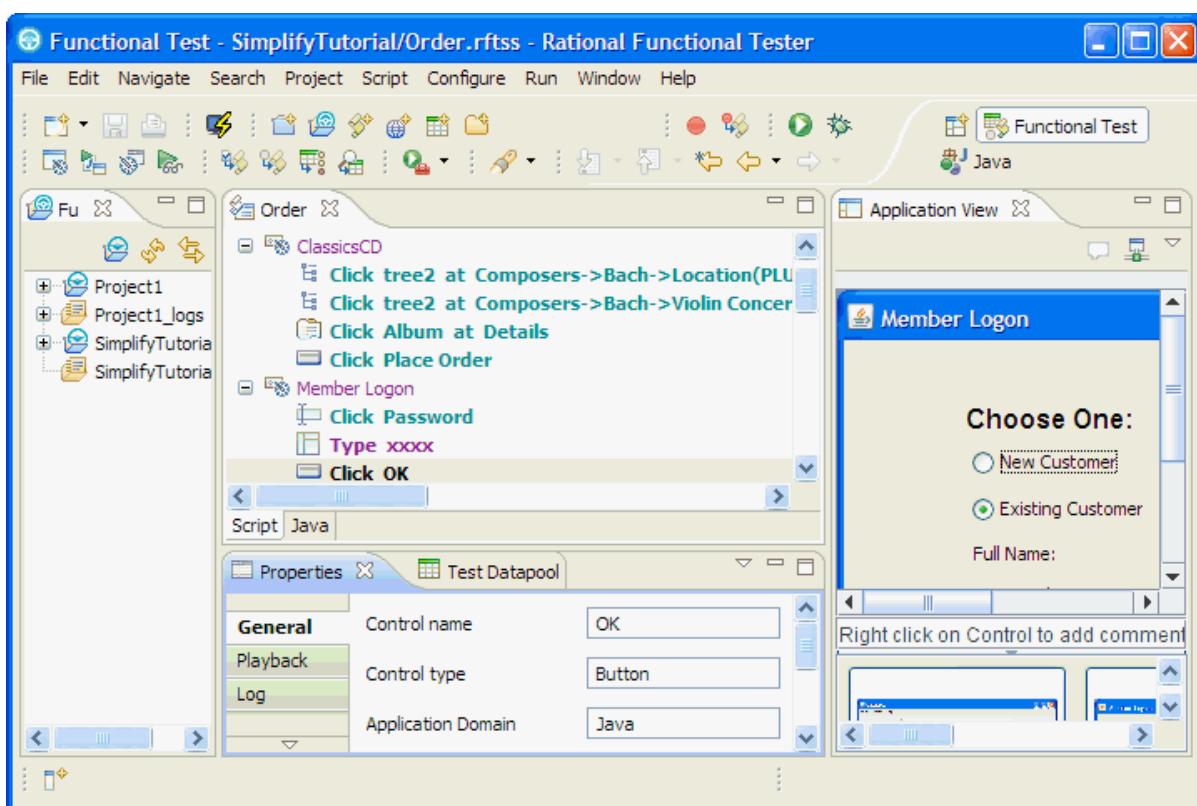
11. Para cerrar la aplicación, pulse el símbolo X que aparece en el ángulo superior derecho de la aplicación ClassicsCD.

Finalización de la grabación

En la barra de herramientas Grabación, pulse Detener grabación (■) para escribir toda la información grabada en el script de prueba.



El script de prueba se muestra en la ventana del editor de script.



Adición de datos al pool de datos

Antes de iniciar el proceso de reproducción, agregue más registros al pool de datos para probar la aplicación de ejemplo ClassicsCD efectuando más pedidos del CD.

El editor del pool de datos tiene el aspecto de esta tabla:

	Item:::java.lan...	_1499:::java.la...	Cantidad:::java...	NumTarjeta:::ja...	TipoTarjeta:::ja...	FechaExpiraci...	Nombre:::j
0	Bach	Violin Concertos	1	9999 9999 999...	Visa	10/11	Trent Culp

1. Coloque el puntero del ratón en el editor de agrupaciones de datos y, a continuación, pulse Intro para añadir una fila después de la primera.
2. Para ahorrar tiempo, copie los datos de la fila 0 de la agrupación de datos en las filas vacías que haya creado.
 - a. Coloque el puntero del ratón en la celda de la fila 0, pulse el botón derecho del ratón y, a continuación, pulse Copiar.
 - b. Coloque el puntero del ratón en la celda de la fila 1, pulse el botón derecho del ratón y, a continuación, pulse Pegar.
 - c. Pulse Sí para pegar los datos en la fila vacía.
3. Cambie el valor de los datos que se encuentran en las columnas **Compositor**, **Artículo**, **Cantidad** e **Importe**.
4. En el editor Agrupación de datos de prueba, pulse X para cerrar el editor de agrupaciones de datos y, a continuación, pulse Sí para guardar los cambios que haya efectuado en la agrupación de datos de prueba.

2.2.2 Reproducción de un script

Al reproducir un script de prueba funcional se repite las acciones registradas y automatiza el proceso de pruebas de software. Con la automatización, puede probar cada nueva construcción de su aplicación más rápido y más bien que por las pruebas de forma manual. Esto disminuye el tiempo de prueba y aumenta la cobertura y consistencia.

Existen dos fases generales en la reproducción de un script:

- En la fase de desarrollo de pruebas, se reproducen scripts para verificar que funcionan tal como se pretendía, utilizando la misma versión de la aplicación que se somete a prueba y que se utiliza para grabar. Esta fase valida el comportamiento que se espera de la aplicación.
- En la fase de pruebas de regresión se reproducen scripts para comparar la última compilación de la aplicación que se somete a prueba con la línea base establecida durante la fase de desarrollo de prueba. La prueba de regresión identifica las diferencias que puedan haber surgido desde la última compilación. Puede evaluar estas diferencias para determinar si se trata de defectos o de cambios.

Para reconocer los objetos de prueba durante la reproducción, RFT utiliza el reconocimiento de pesos de las propiedades del objeto. La siguiente figura explica que durante el proceso de reproducción, RFT puede reconocer un objeto aún si tiene dos propiedades diferentes con respecto a los encontrados en la línea base.

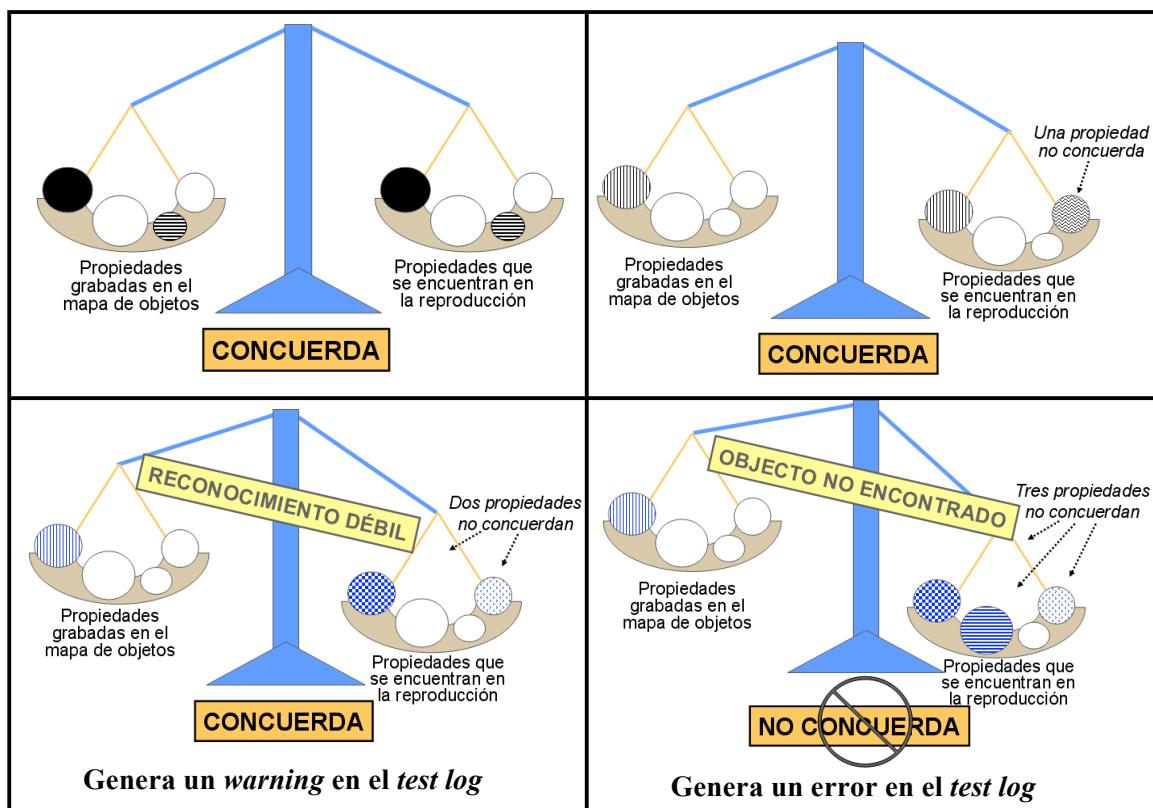


Figura 2.12. Reconocimiento de objetos

Cada propiedad de un objeto en el mapa de objetos de prueba tiene un valor de peso que se asignaron al grabar el script. Este valor de peso es un número de 0 a 100. A cada propiedad de un objeto encontrado durante la reproducción,

RFT le asigna una puntuación de reconocimiento. La mayor puntuación de reconocimiento significa una menor concordancia con la del objeto de la línea base. El objetivo de este mecanismo es permitir la reproducción de scripts a pesar de cambios significativos en los objetos de la aplicación bajo prueba.

2.2.3 Revisión de los resultados

En esta etapa, se analiza el resultado de la prueba a partir del *test log* que se generó al finalizar el proceso de reproducción. Durante la revisión, se determinará la causa de los *warnings* y errores encontrados. Por defecto, el archivo del *test log* se mostrará en formato HTML con 3 secciones en el panel izquierdo: resultado de errores, resultado de *warnings* y puntos de verificación. En la sección de la derecha, se describen con más detalle los errores y *warnings* generados, y un enlace para visualizar la página de Comparación de Puntos de Verificación.

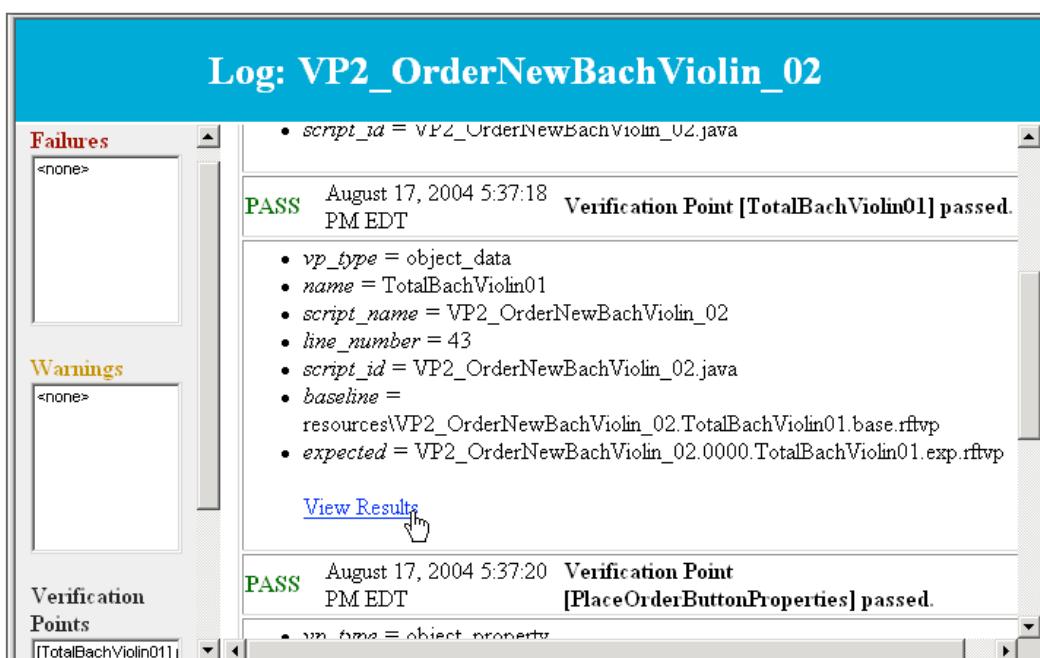


Figura 2.13. Test Log

2.2.4 Características avanzadas de script de pruebas

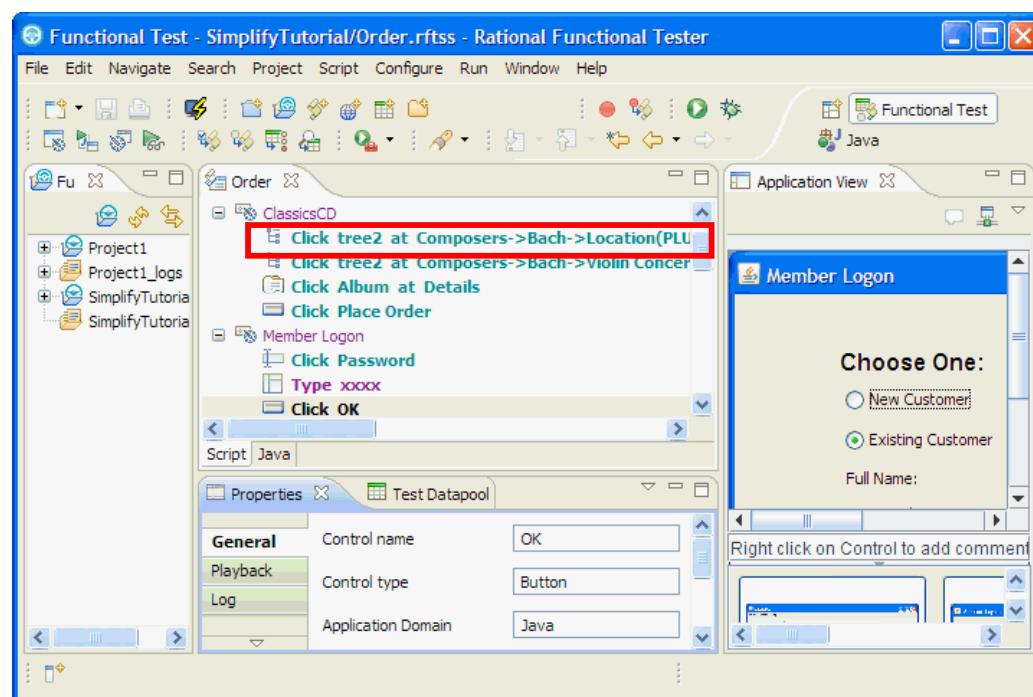
En este apartado, aprenderá a editar un script simplificado mediante la vista de Aplicación.

Los controles de la aplicación, sus datos y sus detalles de propiedades se capturan durante la grabación. Los detalles capturados muestran efectos visuales de la aplicación en la **vista Aplicación**. Puede modificar el script de prueba para probar controles adicionales, o para crear o editar puntos de verificación seleccionando los controles de la aplicación a partir de los efectos visuales de la aplicación sin tener que volver a ejecutar la aplicación bajo prueba.

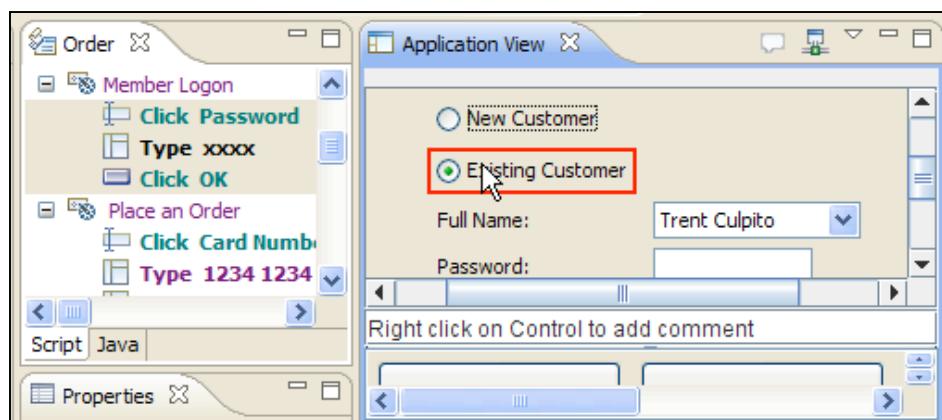
2.2.4.1 Insertar un punto de verificación mediante el efecto visual de la aplicación

Se creará un punto de verificación en el árbol Compositores para comprobar si todos los compositores y sus CD's están listados. Este punto de verificación no se ha insertado cuando se grabó el script de prueba. En lugar de grabar el script o abrir la aplicación para volver a ejecutarla, añada un punto de verificación en el script de prueba desde los efectos visuales de la aplicación. Para ello, realice los siguientes pasos:

1. Desde la pestaña Script seleccione la línea de prueba **Click tree2 at Composers->Bach->Location(Plus_Minus)** (la segunda línea del script de prueba). En la vista Aplicación, el efecto visual de la aplicación resalta la región del árbol Compositores en color azul.



2. Desde la vista de Aplicación, sitúe el ratón encima de la región del árbol de Compositores. La región se resalta en rojo.



- Pulse el botón derecho del ratón y seleccione "Insertar punto de verificación > Punto de verificación de datos".

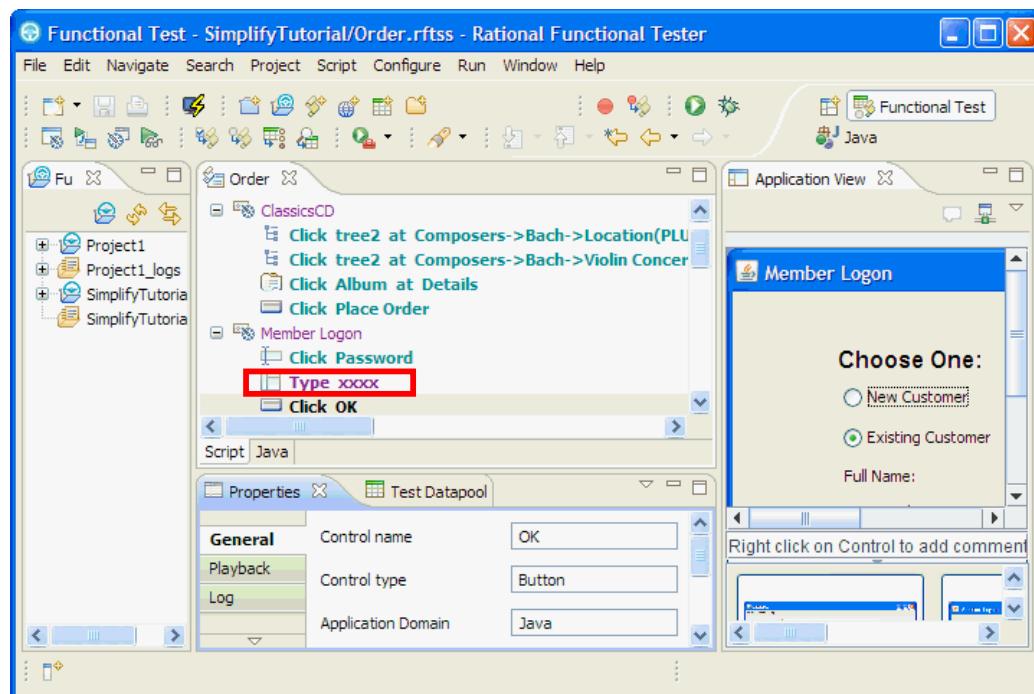
¿Qué ha sucedido en el script de prueba?

La línea de prueba **Verify data in tree2** se añade al editor del script (después de la segunda línea de prueba).

2.2.4.2 Añadir un control adicional a la prueba.

En la ventana Member Logon de la aplicación ClassicsJavaA, no se ha grabado el campo Recordar contraseña para pruebas. Añada el control Recordar contraseña inexistente en el script de prueba:

- Seleccione la línea de prueba Type xxxx en el editor de script. Los efectos visuales de la aplicación Member Logon se muestran en la vista Aplicación.



- Sitúe el ratón encima del control Recordar contraseña en el efecto visual de la aplicación. El control Recordar contraseña se resalta en rojo.
- Pulse el botón derecho del ratón y seleccione el control "Insertar recordar contraseña > seleccionar".

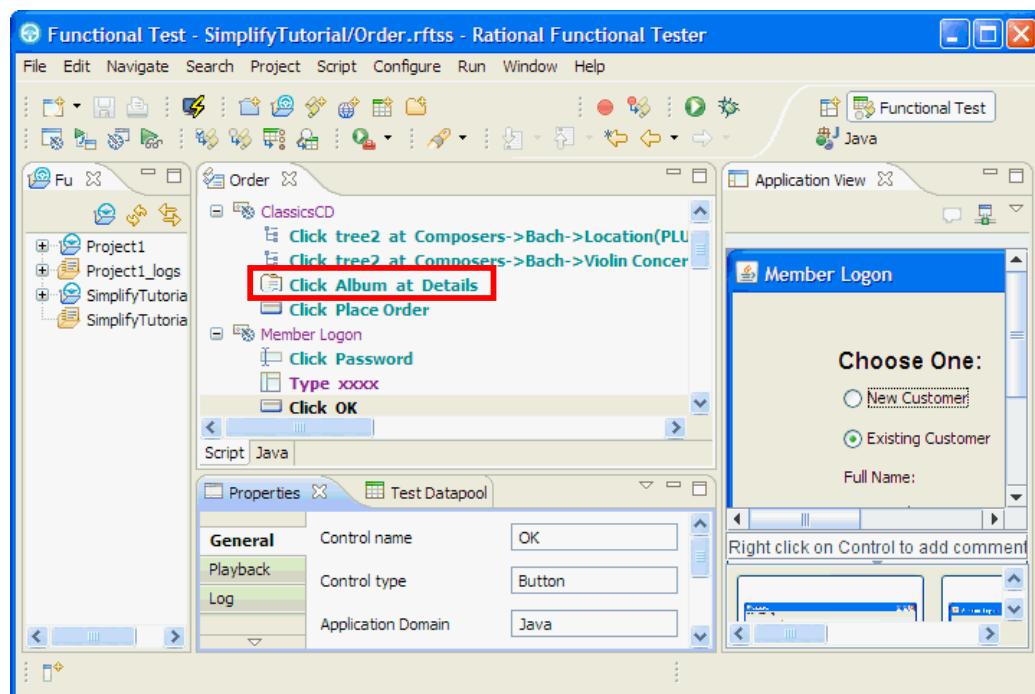
¿Qué ha sucedido en el script de prueba?

La línea de prueba Seleccionar recordar contraseña se inserta en el script de prueba.

2.2.4.3 Cómo modificar la línea de prueba en el editor de script y las propiedades.

Puede modificar la línea de prueba en el editor de script y también especificar detalles tales como los parámetros de reproducción e información de registro para la ejecución de la línea de prueba en la vista Propiedades. Ahora, inhabilite la línea de prueba para ver los detalles del álbum y especificar la información que se debe mostrar en el registro durante la ejecución de la verificación de datos para la lista del control Compositores.

1. Seleccione la línea de prueba Click Album at Details en el editor de script.



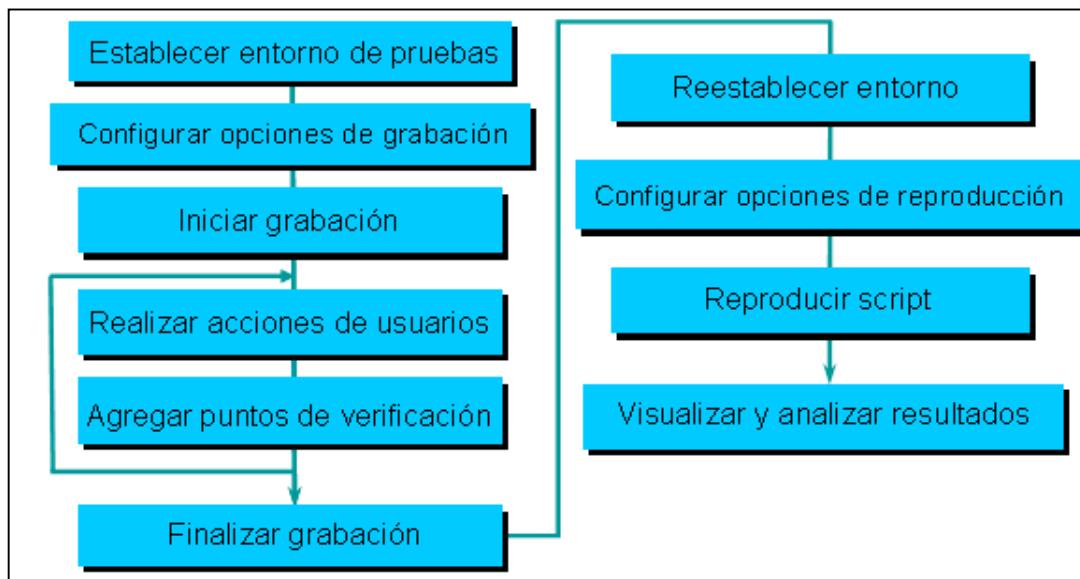
2. Pulse con el botón derecho del ratón y seleccione la acción Habilitar/inhabilitar para inhabilitar la línea de prueba. La línea de prueba no se ejecuta durante la siguiente reproducción.
3. Seleccione la línea de prueba **Verify data in tree2** que se ha insertado en el script mediante el efecto visual de la aplicación.
4. Pulse la página de Registro en la vista Propiedades y seleccione Instantánea de control para ver el estado del control durante la ejecución. Esta instantánea se muestra en el registro de reproducción.

Actividades

1. A partir de la aplicación de escritorio Java entregada en clase, realice los siguientes procesos para un caso de uso:
 - a. Grabación de un script
 - b. Reproducción del script
 - c. Análisis de resultados
2. A partir de la aplicación web Java entregada en clase, realice los siguientes procesos para un caso de uso:
 - a. Grabación de un script
 - b. Reproducción del script
 - c. Análisis de resultados
 - ¿Qué *warnings* se generaron? Comente
 - ¿Qué errores se generaron? Comente
3. A partir de la aplicación web Java de la pregunta 2, realice los siguientes procesos para el mismo caso de uso:
 - a. Modifique la aplicación según indicaciones del docente
 - b. Reproducción del script grabado en la pregunta 2
 - c. Análisis de resultados
 - ¿Qué *warnings* se generaron? Comente
 - ¿Qué errores se generaron? Comente

Resumen

- IBM *Rational Functional Tester* (RFT) es una herramienta automatizada para la realización de pruebas funcionales y de regresión. Permite probar aplicaciones Java, .NET y basadas en Web.
- Con la **tecnología de grabación** del RFT, se puede generar scripts mediante la ejecución y el uso de la aplicación bajo prueba.
- Gracias a la **tecnología ScriptAssure** del RFT se puede crear scripts de prueba más resistentes a los cambios en los objetos de las aplicaciones.
- Un script simplificado de prueba funcional en RFT genera dos archivos: un script con las acciones del usuario y un script con código java.
- Los pasos de alto nivel del proceso de grabación, reproducción y visualización de resultados de un script de prueba funcional se muestra en la siguiente figura:



- En RFT, un punto de verificación captura la información del objeto y los valores literales de la aplicación de prueba y la almacena, en la línea base, a efectos de comparación durante la reproducción. Cuando se reproduce un script, un punto de verificación vuelve a capturar la información del objeto para compararla con la línea base y para ver si se ha efectuado algún cambio, ya sea o no de forma intencionada. Comparar la información del objeto real de un script con la línea base resulta útil para identificar posibles defectos.

Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

☞ <http://publib.boulder.ibm.com/infocenter/rfthelp/v8r1/index.jsp>

En esta página encontrará información del entorno IBM RFT: guía de instalación, guía de aprendizaje, entre otros temas.

☞ <http://www.ibm.com/developerworks/ssa/rational/library/09/functionaltestertableobjects/#download>

Este artículo muestra cómo usar IBM RFT para automatizar objetos ajustados dentro de las celdas de una tabla HTML manteniendo las celdas especificadas como referencia. Lo ayudará a automatizar operaciones con estos objetos.

☞ <http://www-01.ibm.com/support/docview.wss?uid=swg21213488>

En esta página se describe la gestión de los objetos de prueba en el IBM RFT.



FUNDAMENTOS RATIONAL PERFORMANCE TESTER

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, el alumno realiza pruebas de rendimiento para su proyecto final, el cual permite resolver los desafíos de pruebas de rendimiento más comunes.

TEMARIO

3.1. Tema 5: Introducción al Rational Performance Tester

- 3.1.1. Arquitectura de Rational Performance Tester
- 3.1.2. Características y beneficios de Rational Performance Tester

3.2. Tema 6: Pruebas de rendimiento en RPT

- 3.2.1. Grabación de pruebas
- 3.2.2. Programación de pruebas
- 3.2.3. Ejecución de pruebas de rendimiento

ACTIVIDADES PROPUESTAS

- Los alumnos realizan pruebas de rendimiento para un proyecto web Java.

3.1. INTRODUCCIÓN AL RATIONAL PERFORMANCE TESTER

Las pruebas de rendimiento tienen un significado diferente para diferentes personas, pero en general, estas pruebas se realizan para:

- Determinar el tiempo de respuesta de una solicitud
- Determinar el número de usuarios que el sistema soportará
- Determinar la configuración óptima del sistema
- Averiguar qué sucede cuando un sistema está sometido a carga pesada

Una prueba de rendimiento es el proceso de ejecución de un sistema mediante la actividad de emulación con el fin de:

- Observar el comportamiento del sistema
- Recolectar datos y métricas

Más allá de la observación y la recolección, los ingenieros utilizan los datos y métricas para mejorar el rendimiento de la aplicación objetivo. Este descubrimiento y la resolución de los cuellos de botella (los casos de bajo rendimiento o inaceptable) es el retorno real de la inversión (ROI) de una herramienta de pruebas de rendimiento.

Las actividades típicas de emulación son:

- Transacciones del usuario y escenarios de los CU
- Períodos de mayor actividad
- Trabajos por lotes y otros procesos internos del sistema
- Administrador de tareas (por ejemplo, copias de seguridad del sistema)

Los objetivos de las pruebas de rendimiento son:

- Determinar los tiempos de respuesta del sistema
- Determinar el número máximo de usuarios de un sistema (componentes, transacción, o configuración)
- Descubrir las configuraciones óptimas o mínima del sistema

Las pruebas de rendimiento pueden ayudar a reducir los costes del sistema, determinando qué recursos del sistema, tales como los re cursos de memoria y disco, son necesarios para ofrecer un rendimiento aceptable. Comprender los requisitos mínimos permite tomar mejores decisiones sobre qué hardware y software debe ser comprado para el despliegue de la aplicación objetivo.



Figura 3.1. Costo por Defecto en Diseño, Pruebas y Producción

Existen muchas herramientas para realizar pruebas de rendimiento, Rational Performance Tester (RPT) es una de ellas. RPT es una solución de verificación de cargas y rendimiento para equipos que se ocupen de la capacidad de ampliación de sus aplicaciones basadas en web. Gracias a la combinación de funciones de análisis detallados y fáciles de utilizar, Rational Performance Tester simplifica la creación de pruebas, la generación de cargas y la recopilación de datos para garantizar que las aplicaciones se amplíen hasta miles de usuarios concurrentes.

3.1.1 Arquitectura de Rational Performance Tester

3.1.1.1 Relación entre el *workbench* y *workspace*

El *workbench* es la interfaz de usuario y el entorno de desarrollo integrado (IDE) en Eclipse y en los productos de la plataforma de desarrollo de software IBM Rational que se basan en Eclipse. Mientras que el *workspace* es la colección de proyectos y otros recursos que están actualmente en desarrollo en el *workbench*.

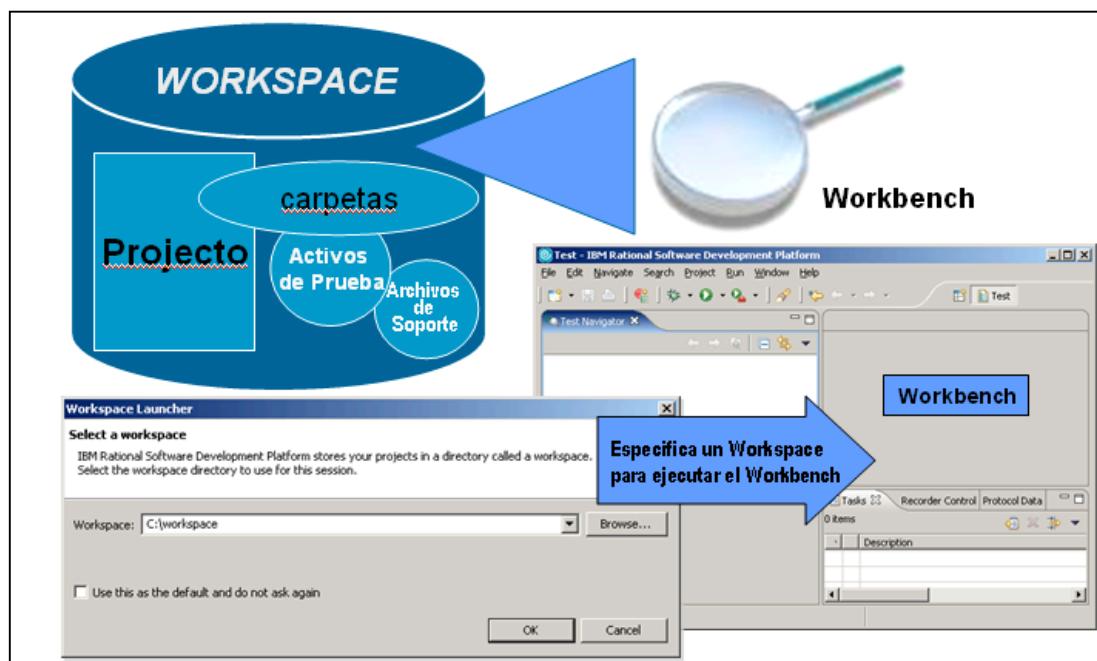


Figura 3.2. Relación entre el *workbench* y *workspace*

3.1.1.2 Perspectiva de Prueba en RPT.

La siguiente figura muestra el entorno del RPT con la perspectiva **Performance Test** activa.

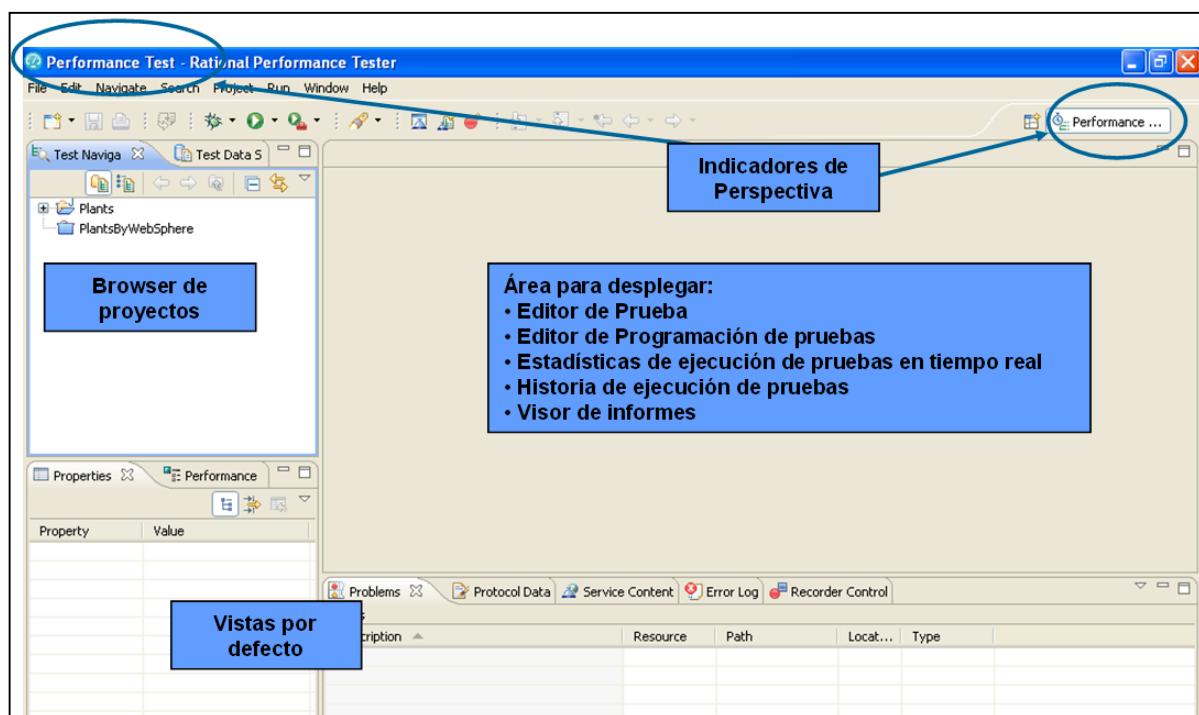


Figura 3.3. Perspectiva **Performance Test**

Editor de Prueba

Al finalizar la sesión de grabación de un escenario de caso de uso se muestra el editor de pruebas con la información de las páginas visitadas.

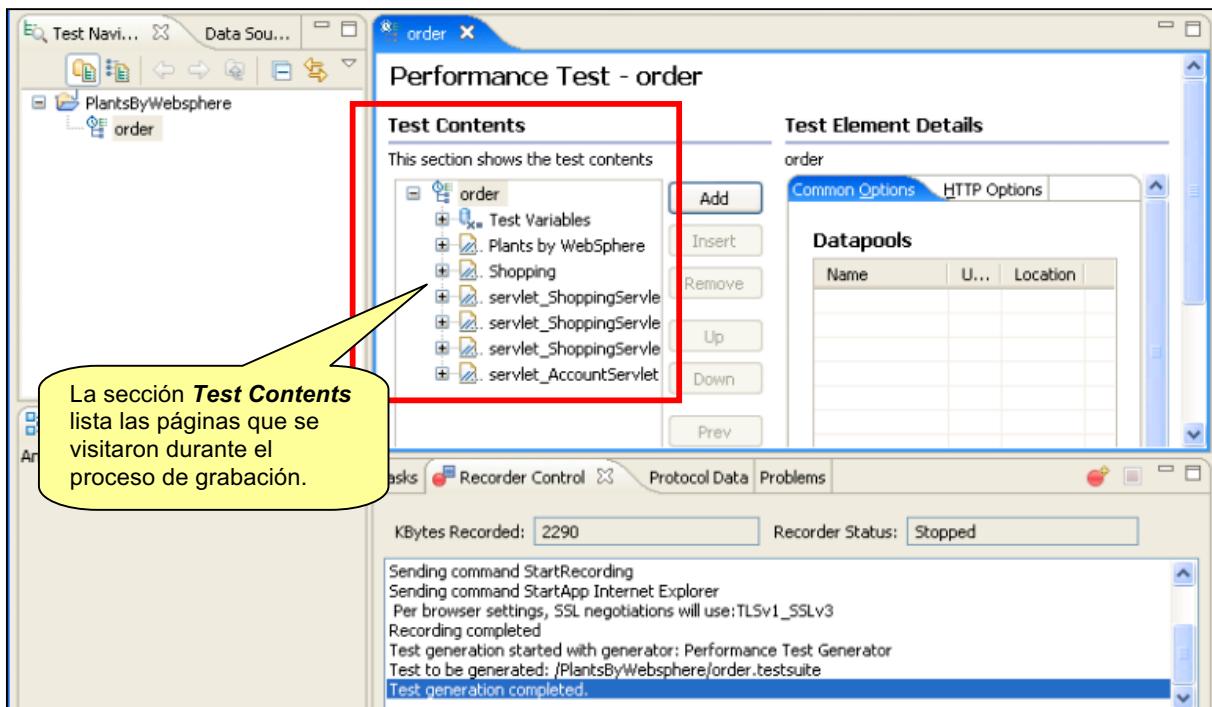


Figura 3.4. Vista de la prueba al finalizar el proceso de grabación

Al seleccionar una página de la sección **TestContents**, en la sección **Test Elements Details** se muestra su información detallada.

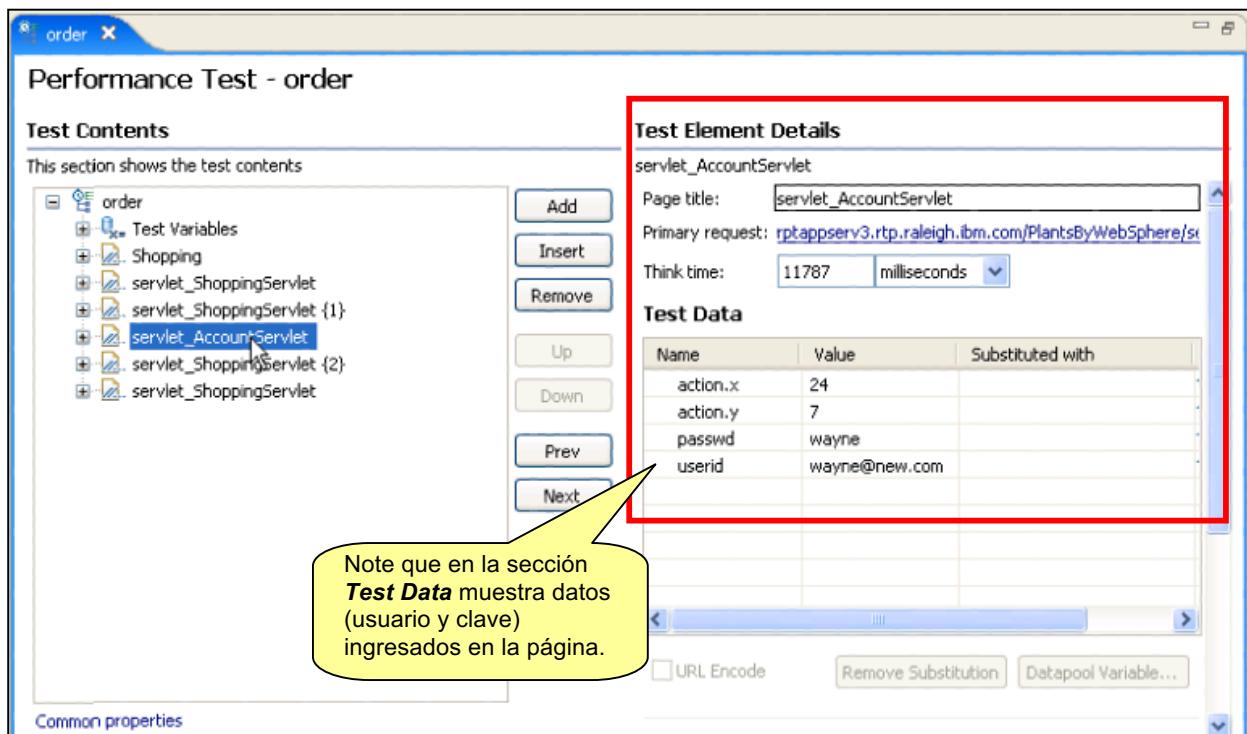


Figura 3.5. Información detallada de una página de prueba

Al desplegar una página de prueba, se visualiza las solicitudes que se generaron desde esta página.

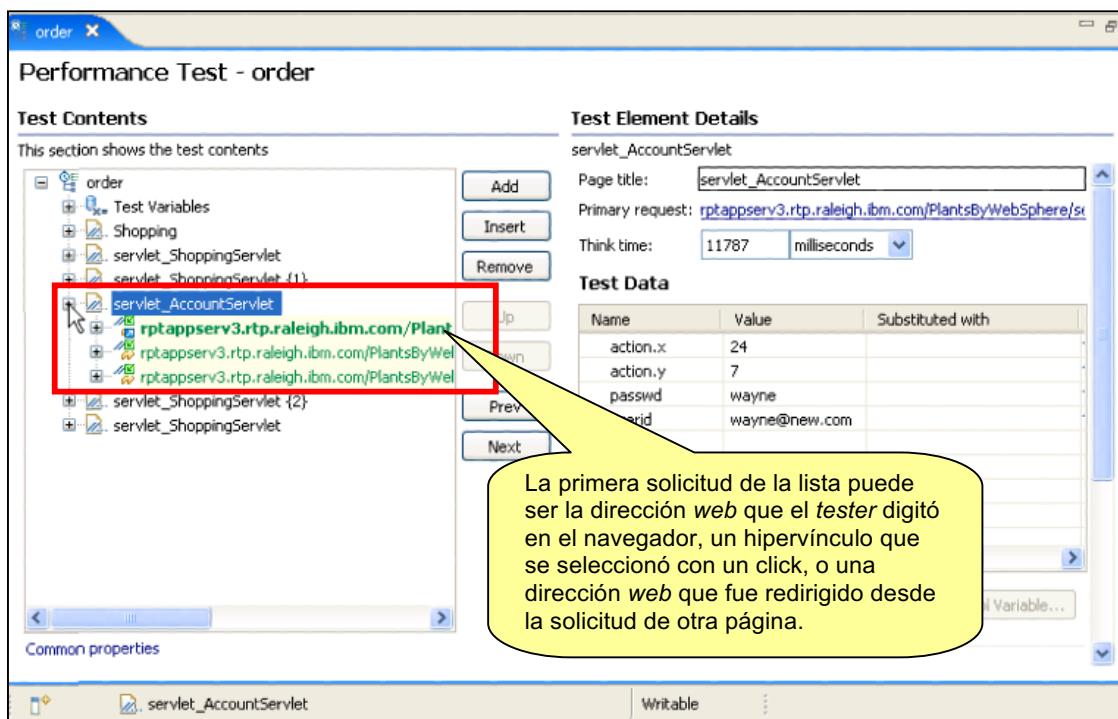


Figura 3.6. Solicitudes de la página de prueba

También se puede visualizar la respuesta generada para la solicitud de una página.

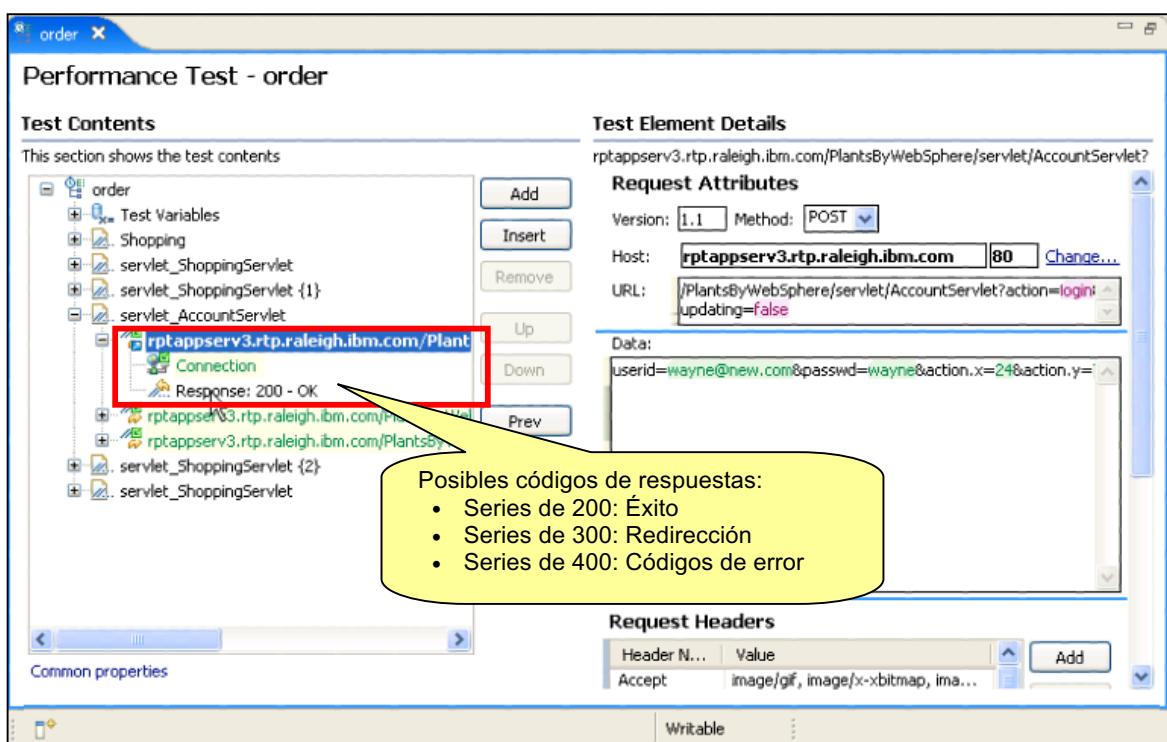


Figura 3.7. Respuesta de la solicitud seleccionada

Editor de Programación de Pruebas

En el editor de programación de pruebas se diseña la carga de trabajo a fin de emular con precisión las acciones de los usuarios individuales.

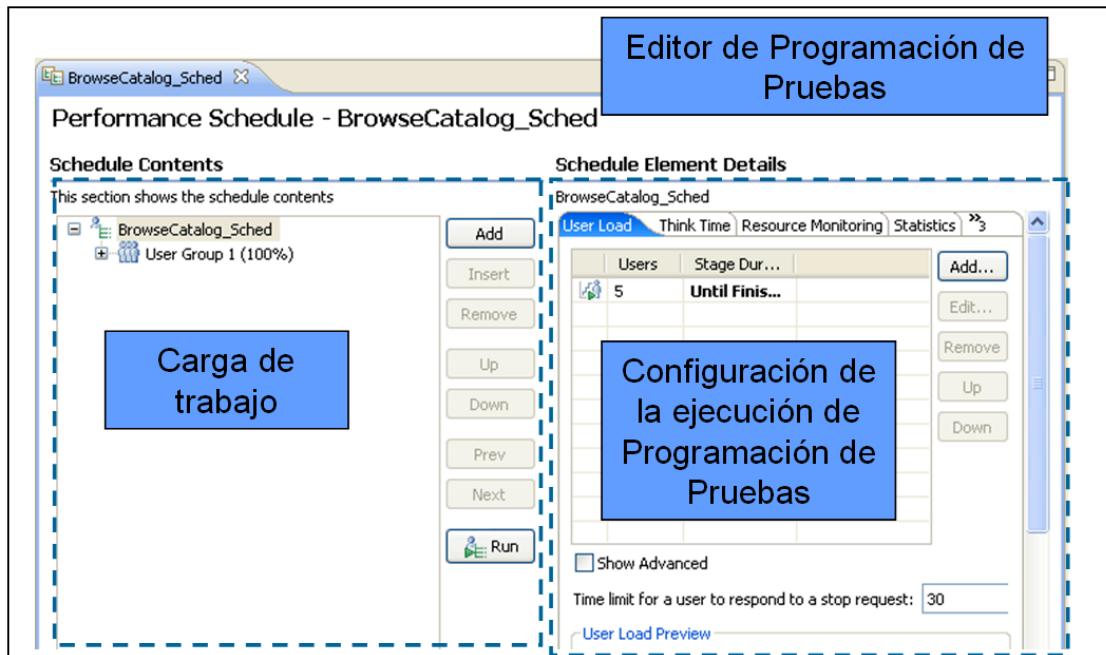


Figura 3.8. Editor de programación de Pruebas

Estadísticas de Ejecución de Pruebas en Tiempo Real

La vista **Performance Test Runs** muestra los resultados de la prueba y es donde se puede acceder para visualizar los informes. Esta vista también controla los siguientes elementos durante la ejecución de la prueba:

- Nivel de *Log*: Es un indicador de la cantidad de registros por hacer. Cuanto más alto sea el nivel, mayor información será registrada.
- Usuarios virtuales: Agregar o restar el número de usuarios virtuales durante la ejecución de la prueba.
- Detener ejecución de la prueba: Detener la prueba.

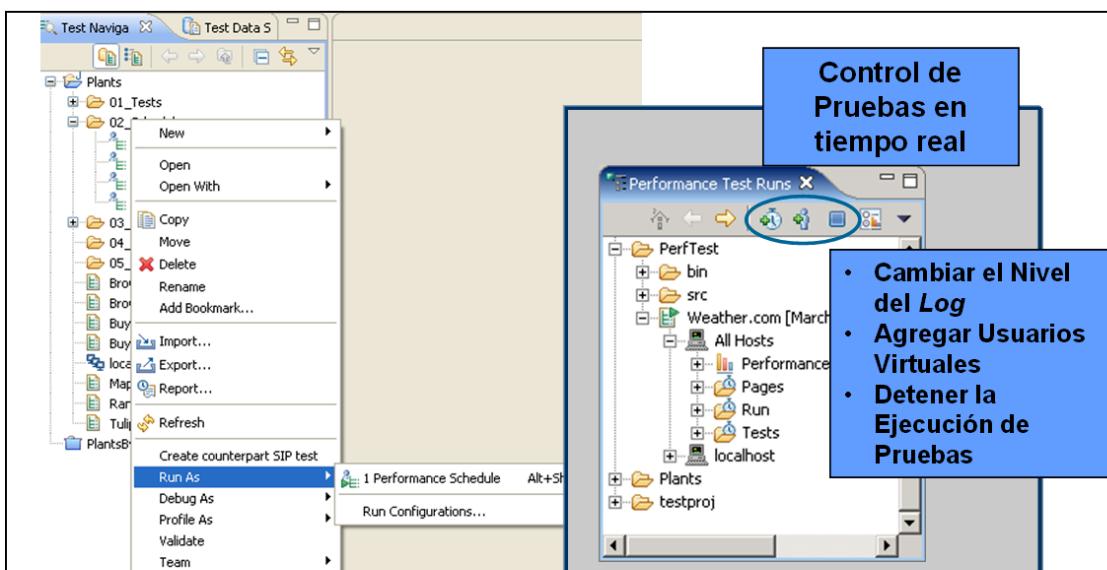


Figura 3.9. Vista Performance Test Runs

Los informes en tiempo real están disponibles durante la ejecución de la prueba. Son idénticos a los informes por defecto que se pueden acceder después de la ejecución de pruebas. Puede seleccionar una de las pestañas, que contienen diversos informes, desde la parte inferior del monitor de pruebas. Tal como se indica en la siguiente figura.

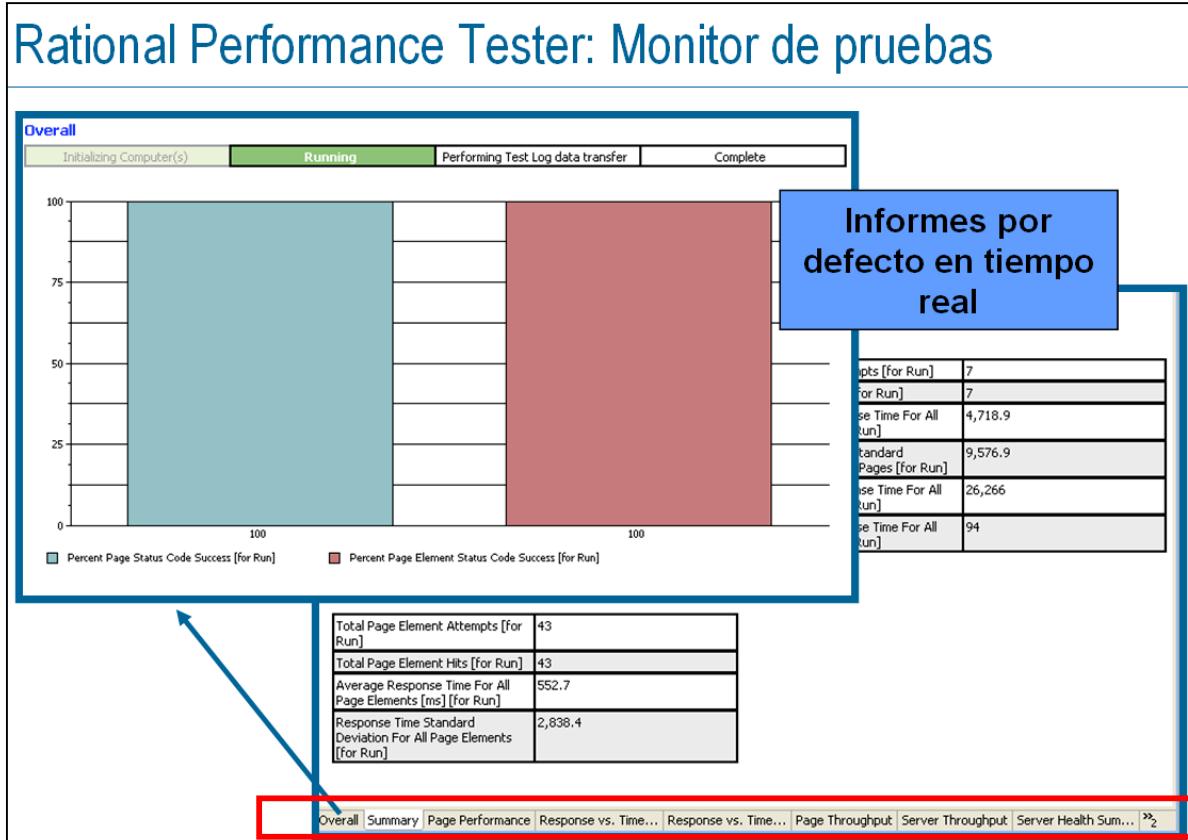


Figura 3.10. Monitor de pruebas durante el proceso de ejecución

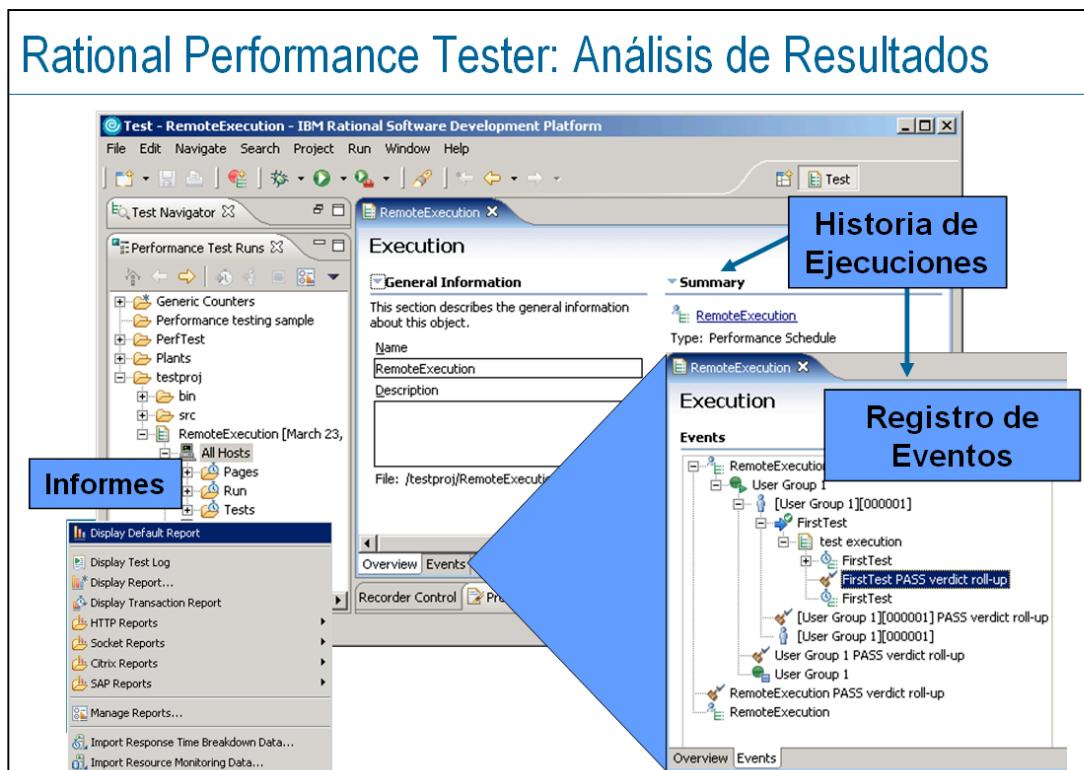


Figura 3.11. Informe de resultados después del proceso de ejecución

3.1.2 Características y beneficios de Rational Performance Tester

A continuación se listan las características y los beneficios conseguidos gracias a las características del RPT.

- **Múltiples técnicas, poco intrusivas de grabación capturan la comunicación cliente servidor involucrando los protocolos tanto HTTP/HTTPS como SQL:** La grabación del script ocurre en forma no obstructiva mientras el *tester* interactúa con la interfaz de usuario de la aplicación cliente. La grabación de la comunicación resultante permite la coexistencia de múltiples protocolos en un único script - en lugar de tener que fragmentarlo en grabaciones de cada protocolo en scripts separados - simplificando la documentación y modificación del test.
- **Los filtros incluidos de correlación de datos detectan datos variables y preparan los tests para la generación de de carga test basados en los datos:** Para asegurar la equivalencia de las cargas de usuario simulados a las del mundo real, RPT contiene un utilitario de correlación de datos que detecta automáticamente los datos variables usados durante el proceso de grabación. Mediante su "*datapool wizard*", los *testers* de la performance pueden generar datos variables (como nombres, números de tarjetas de crédito, direcciones, etc) los mismos serán accedidos durante la ejecución del test, asegurando que cada usuario simulado presente datos únicos el ambiente servidor. Sin este paso, los tests de performance no reproducirían el ambiente de producción fallando en consecuencia al tratar de exponer los verdaderos problemas de escalabilidad.
- **Un rico lenguaje y una librería de funciones de test permiten a los testers expertos una extensa customización de los scripts que requieren de algoritmos avanzados de análisis:** El test script del **Performance Tester** usa un lenguaje poderoso, similar al C que permite a los *testers* avanzados de performance el implementar una variedad ilimitada de técnicas de personalización para facilitar análisis y manipulación de los datos durante la ejecución del test. Tiene incluidos un conjunto de funciones predefinidas particularmente útiles para técnicas avanzadas de *testing* de carga.
- **Tiene un scheduler de carga de test que puede ser personalizado completamente para crear modelos de perfiles reales de carga de usuarios altamente precisos:** Antes que se pueda ejecutar el test, los *testers* de performance deben determinar con precisión el modelo de los diversos grupos de usuarios que usarán el sistema en la realidad. **Rational Performance Tester** contiene un *scheduler* avanzado de carga de trabajo para permitir este modelado, facilitando que los *testers* alcancen a replicar el mundo real de los usuarios finales. Están disponibles variadas construcciones como *loops* y saltos condicionales y los múltiples perfiles de usuario resultantes se pueden ejecutar en paralelo.
- **Reportes en tiempo real muestran segundo a segundo los tiempos de respuesta de usuarios y grupos de usuarios exponiendo así los cuellos de botella del sistema tan pronto**

como ocurren: Durante la ejecución del test, RPT muestra gráficamente vistas consolidadas de los tiempos promedio de respuesta - a lo largo del tiempo - para cada perfil de usuario simulado. Además, los *testers* tienen la opción de profundizar en la conversación en tiempo real entre cualquier instancia de usuario individual y el sistema bajo test para ver los datos intercambiados, permitiendo un análisis rápido y temprano de la degradación sospechada de la respuesta. Se dispone de reportes adicionales pos-ejecución del test para un posterior análisis de los datos respondidos - todos exportables a archivos .CSV y HTML.

- **Se pueden adquirir y correlacionar los datos detallados de los recursos servidor con los tiempos de respuesta del mismo para exponer limitaciones de performance relacionadas al hardware:** Una variedad de métricas de recursos del servidor son reunidas por el RPT durante la ejecución del test. Estas métricas se pueden mostrar lado a lado con los datos de respuesta reunidos por el servidor para analizar si la pobre performance del sistema se puede atribuir a las características del hardware (tales como el tiempo de encolado y memoria del sistema) más que al software de aplicación.

3.2. PRUEBAS DE RENDIMIENTO EN RPT

Existen muchos tipos de pruebas de rendimiento, por ejemplo:

- De carga: Determina los tiempos de respuesta de todos los procesos de negocio críticos y las transacciones importantes de la aplicación. Esta prueba puede mostrar el número esperado de usuarios concurrentes utilizando la aplicación y que realizan un número específico de transacciones durante el tiempo que dura la carga.
- De volumen: Determina el rendimiento y capacidad de *drivers* asociados a un proceso o transacción específica.
- De stress: Determina la solidez de la aplicación en los momentos de carga extrema y ayuda a los administradores para determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.

Las pruebas de rendimiento con RPT pueden ser automatizadas. Para ello, debe grabar los escenarios a probar, programar las pruebas y ejecutarlas a fin de analizar los informes obtenidos según las necesidades del equipo. Estos procesos se detallan a continuación con un caso práctico.

CASO PRÀCTICO: DESARROLLO DE UNA PRUEBA DE RENDIMIENTO

La aplicación sobre la cual se realizarán las pruebas de rendimiento es PLANTS by WebSphere.

Plants by WebSphere, es una aplicación e-commerce basada en Web que da soporte a la venta de semillas, plantas y herramientas de jardinería.

1. Preparación para la grabación de las pruebas

1. Preparar el entorno de pruebas

- Configurar Herramientas de prueba
- Configurar entorno para la ejecución de la aplicación objetivo

2. Grabar pasos detallados del usuario para la prueba: Registrar una orden

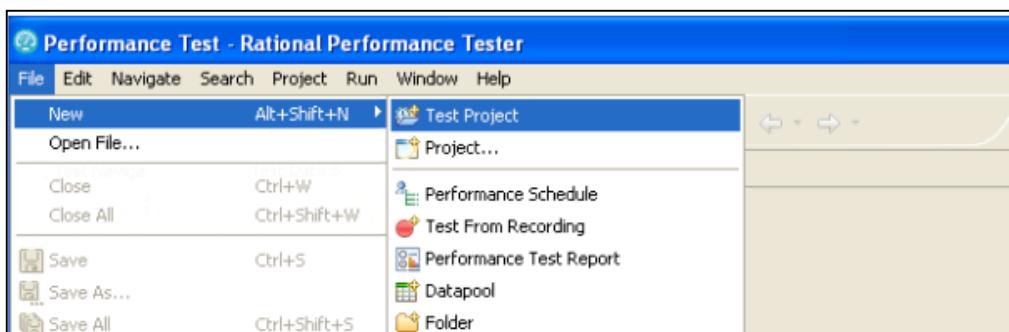
1. Página de Inicio
 - Seleccione Bonsai Tree
2. Página de Descripción del producto
 - Pulse ADD to CART
3. Página de Shopping cart
 - Confirme 1 árbol Bonsai
 - Pulse Checkout now
4. Página de Inicio de Sesión
 - Ingresar la cuenta de email por default y el password
 - Pulse en Sign in
5. Página de Checkout
 - Especifique el embarque y la información de facturación
 - Active el checkbox para especificar que se use la información de la cuenta por defecto y la dirección de facturación
 - Ingrese el número de la tarjeta de crédito 1234 1234 1234
 - Cambie el año de expiración a 2011
 - Pulse continue
6. Revise la página de órdenes
 - Pulse Submit Order
7. Página Order Confirmation
 - Confirme que el número de orden se ha incrementado

2. Grabación de las pruebas

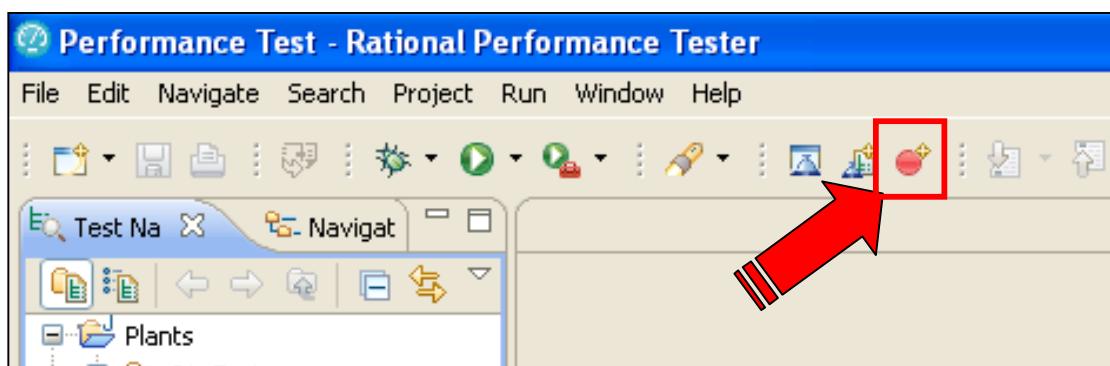
El siguiente gráfico muestra el proceso general:



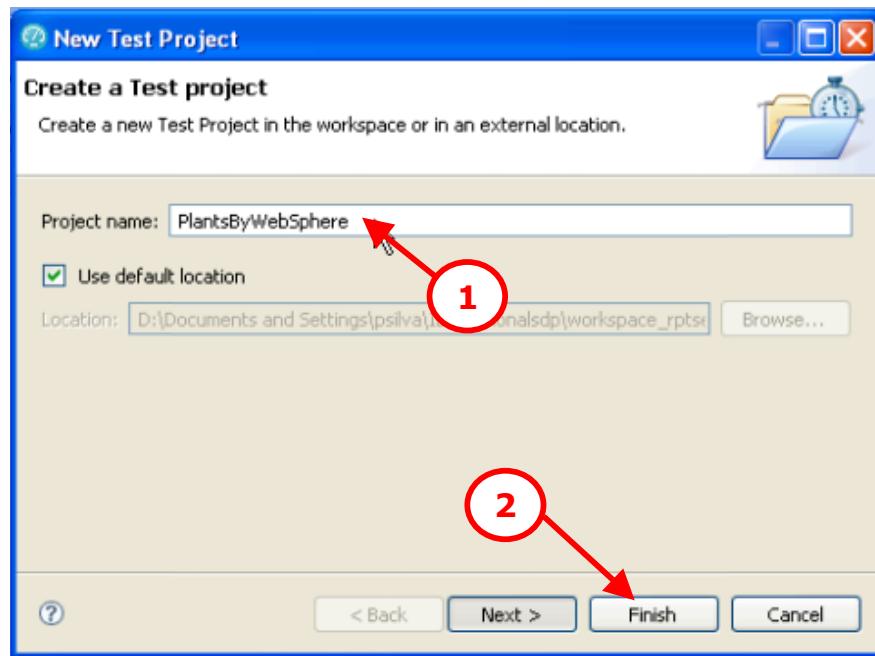
1. Seleccione **File > New > Test Project** para crear el proyecto que contendrá las pruebas de la aplicación.



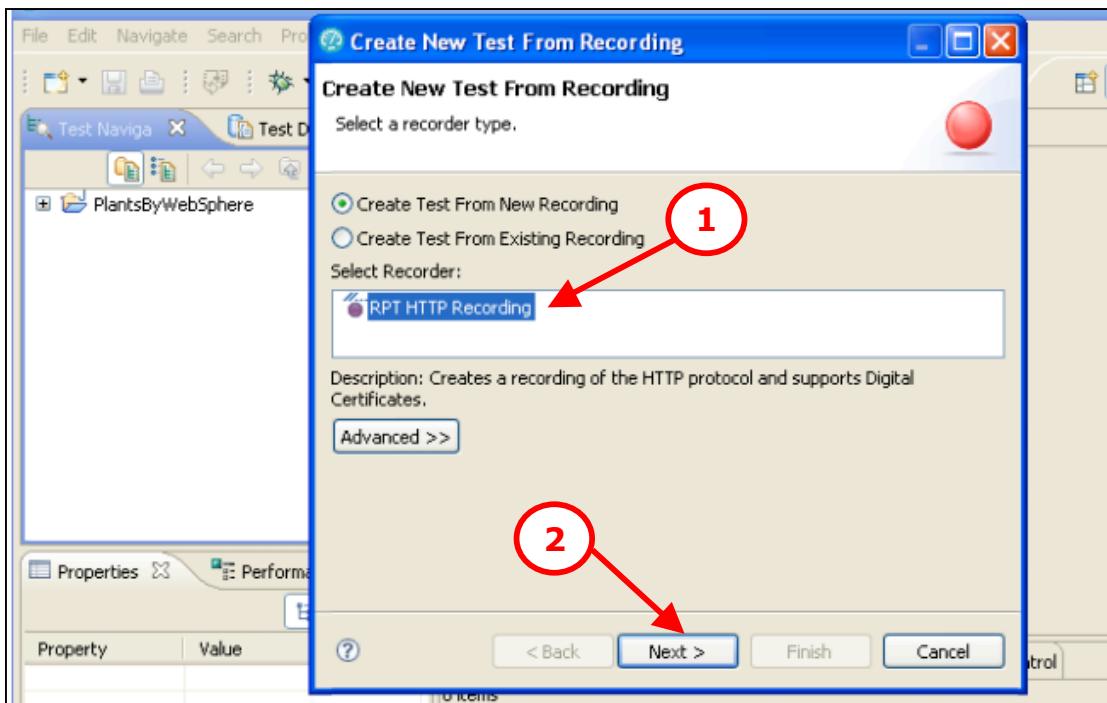
También puede crear el proyecto al seleccionar la opción de grabación desde la barra de herramientas:



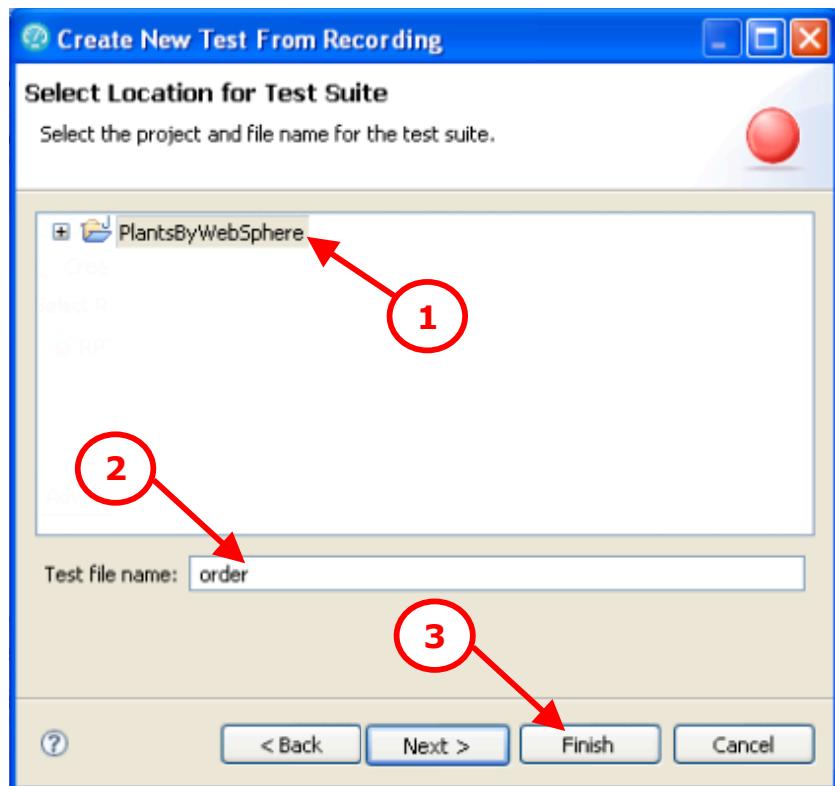
2. Desde la ventana “Nuevo Proyecto de Prueba” edite el nombre del proyecto y luego seleccione **Finish**.



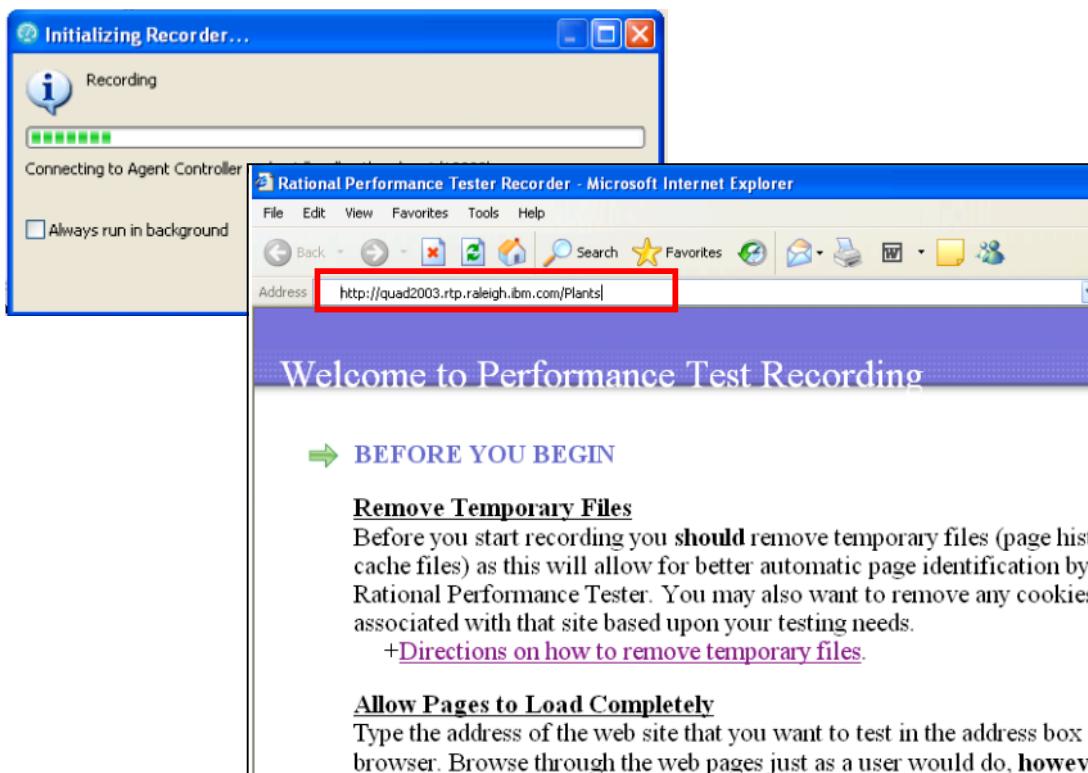
3. Note que en el explorador de pruebas se ha creado su proyecto. Seleccione **RPT http Recording** para iniciar la grabación, luego **Next**.



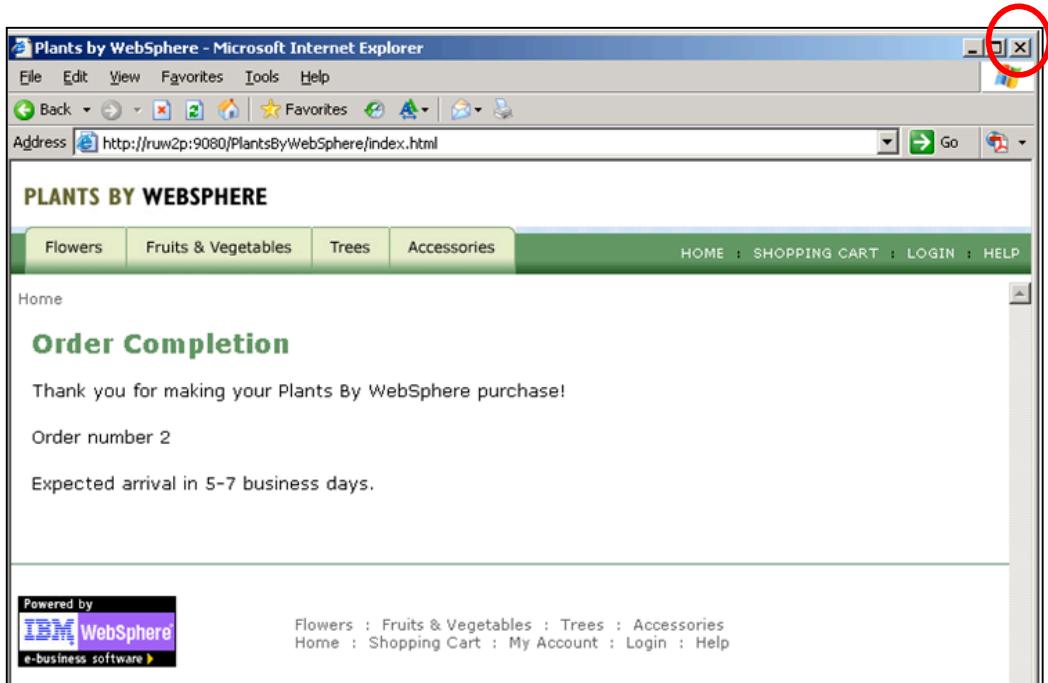
4. Ahora, seleccione el proyecto, edite el nombre de su primera prueba y seleccione **Finish**.



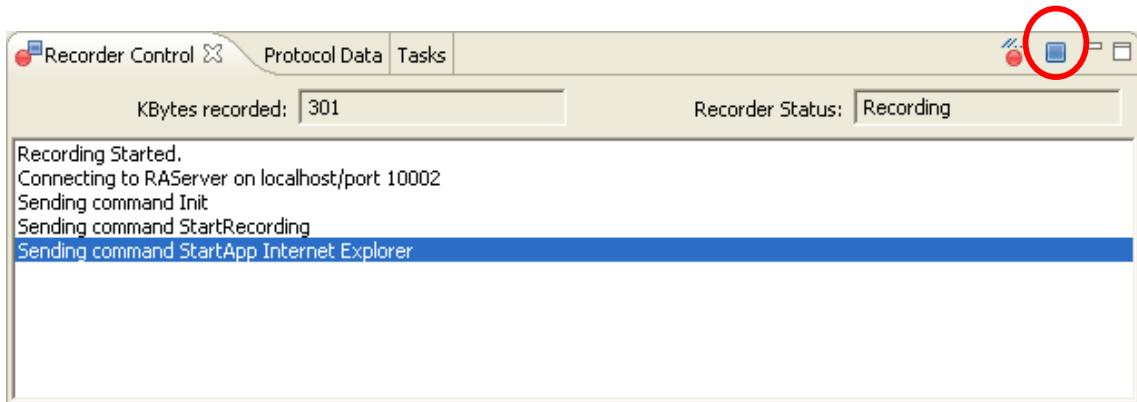
5. A continuación, se mostrará el indicador de inicio de grabación y luego el browser de Internet en donde escribirá el URL de su aplicación y **Enter** para empezar a grabar.



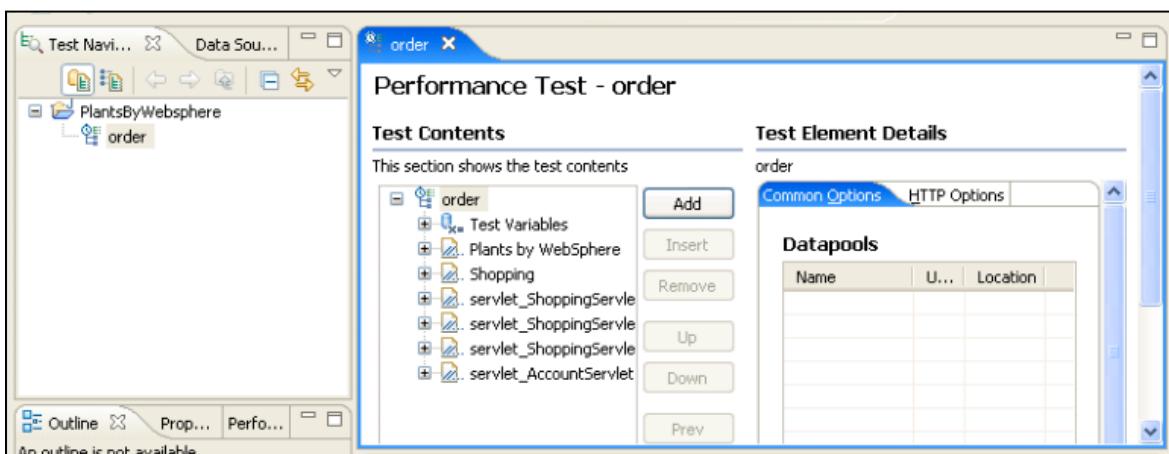
6. Después de interactuar con su aplicación para grabar el escenario “**Registrar una orden**”, cierre el browser de Internet para detener el proceso de grabación.



También puede detener el proceso de grabación desde la opción **Stop Recording** de la vista **Recorder Control**.



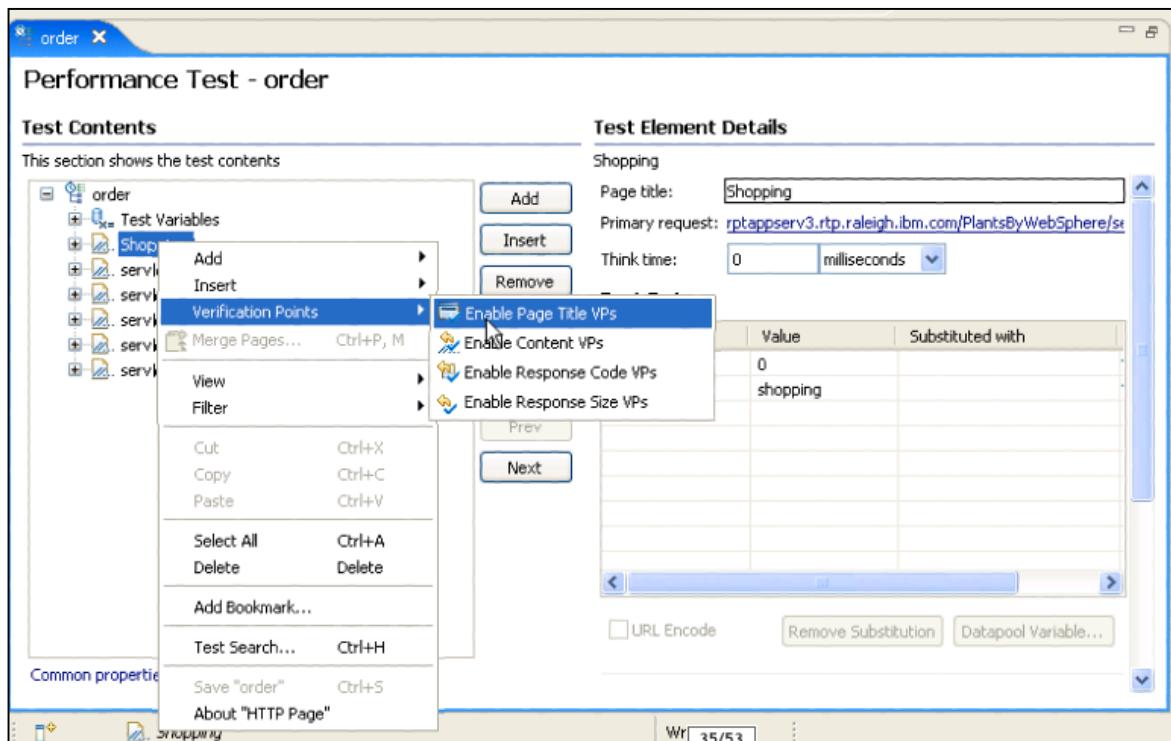
7. Desde el editor de pruebas podrá visualizar la prueba generada.



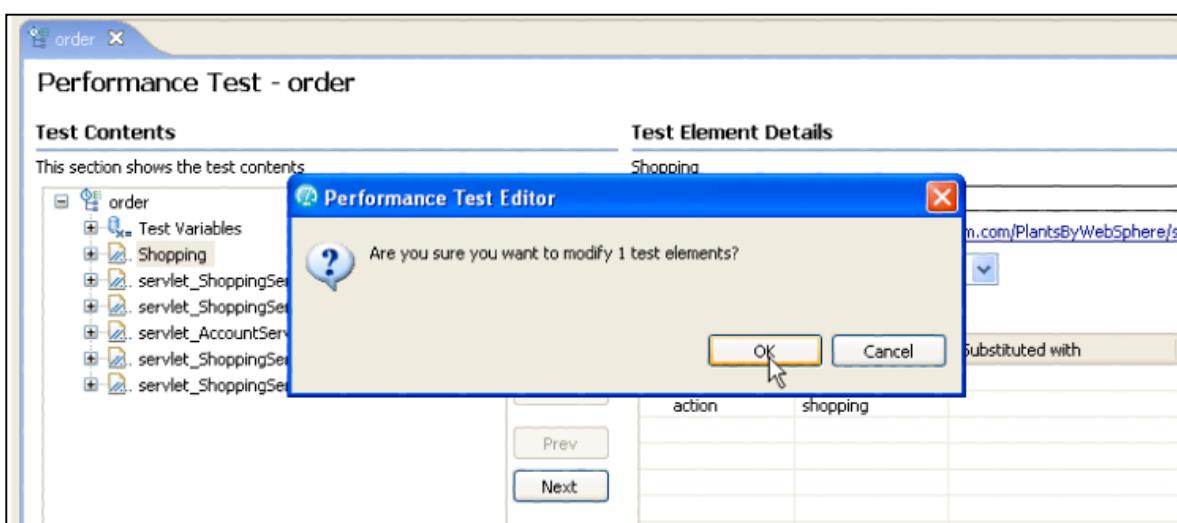
3. Edición de Pruebas

En este punto, aprenderá a añadir puntos de verificación y pool de datos a una prueba. Los puntos de verificación comprueban si se da el comportamiento esperado durante una ejecución. El pool de datos permite variar los valores de lo registrado en la prueba.

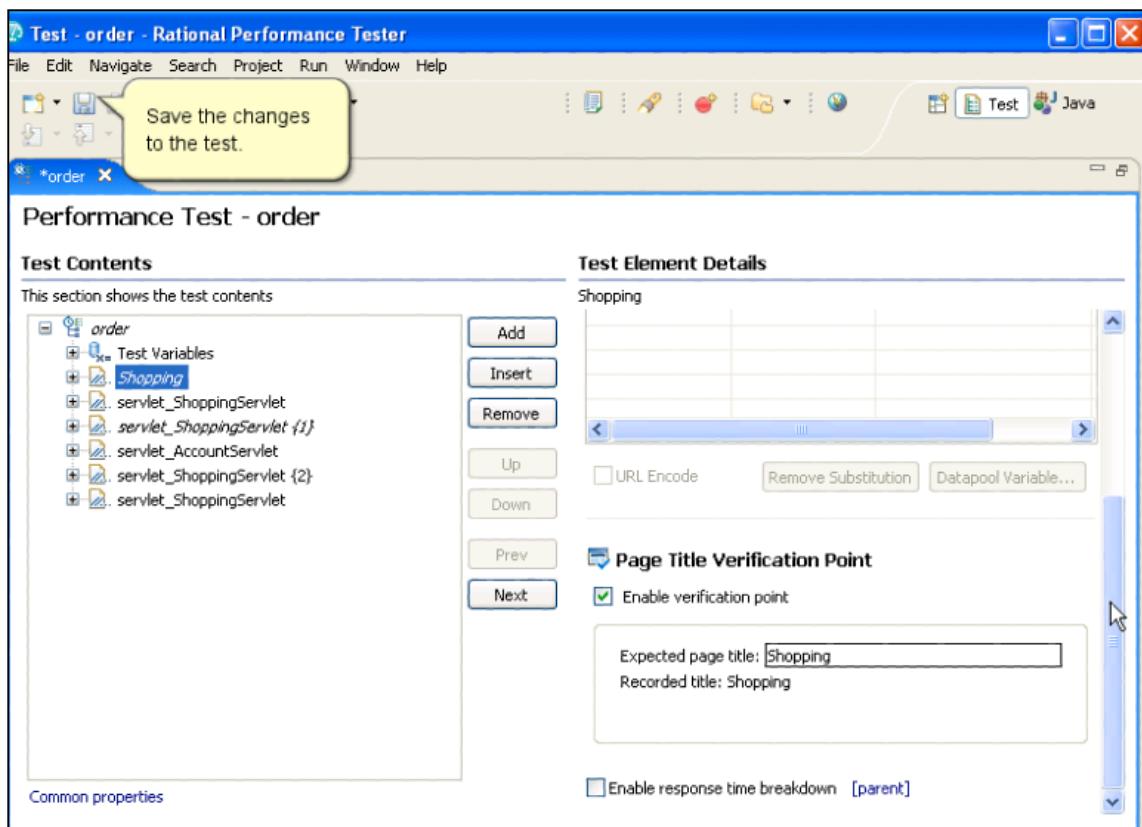
1. Click derecho sobre la página a la cual agregará un punto de verificación (PV) > **Verification Points > Enable Page Title VPs.**



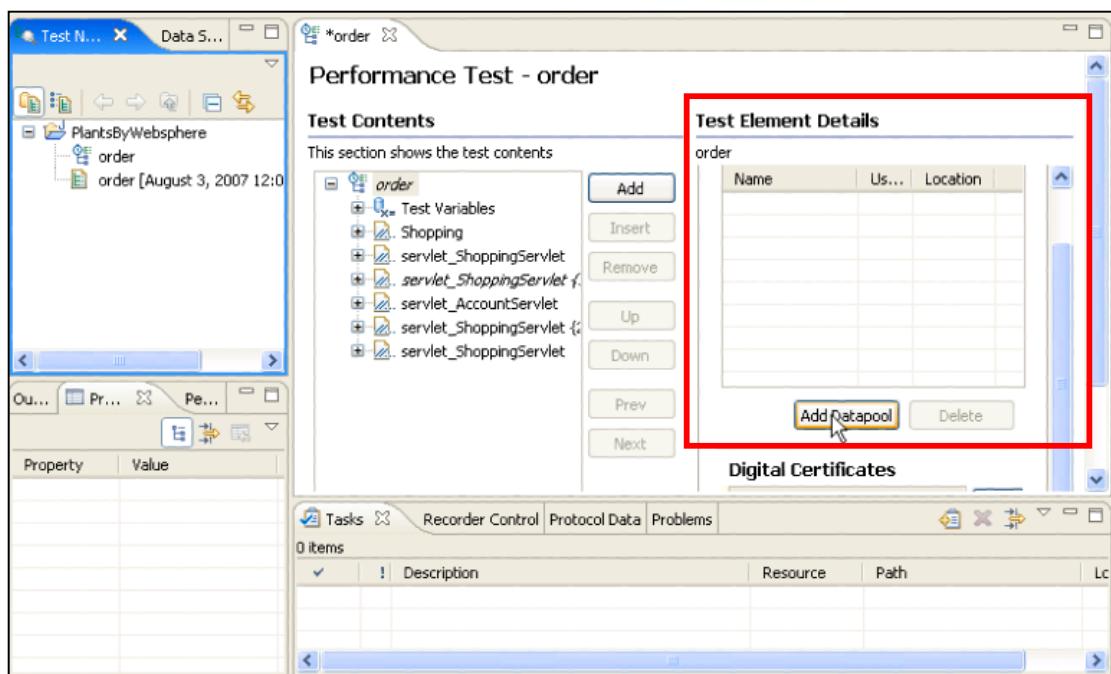
2. Confirme la agregación del PV.



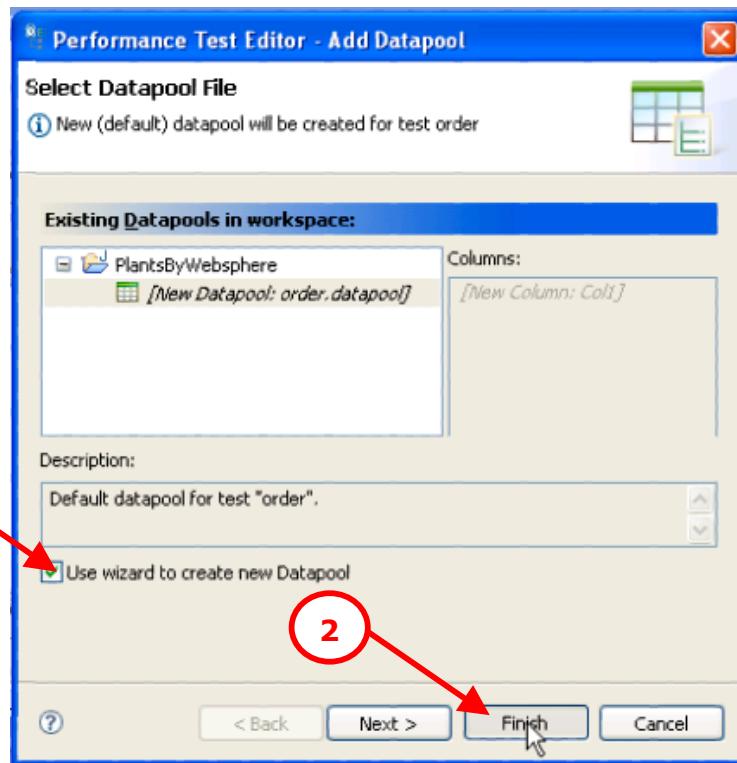
3. A continuación, se mostrará la sección del PV agregado; guarde los cambios.



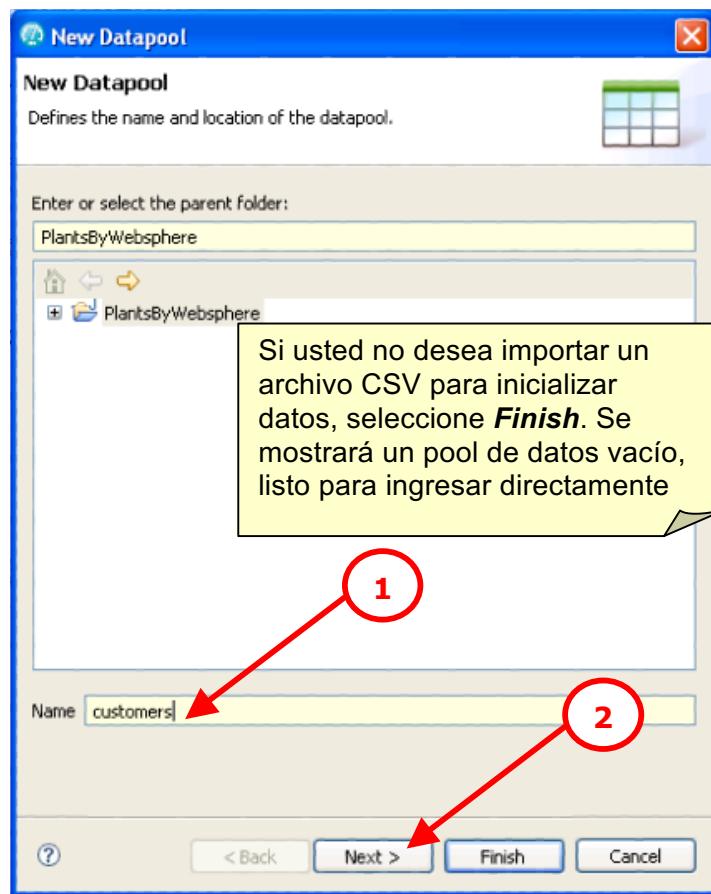
4. Ahora, importará un archivo CSV para crear un pool de datos. Para ello, desde la sección **Datapools** seleccione **Add Datapool**.



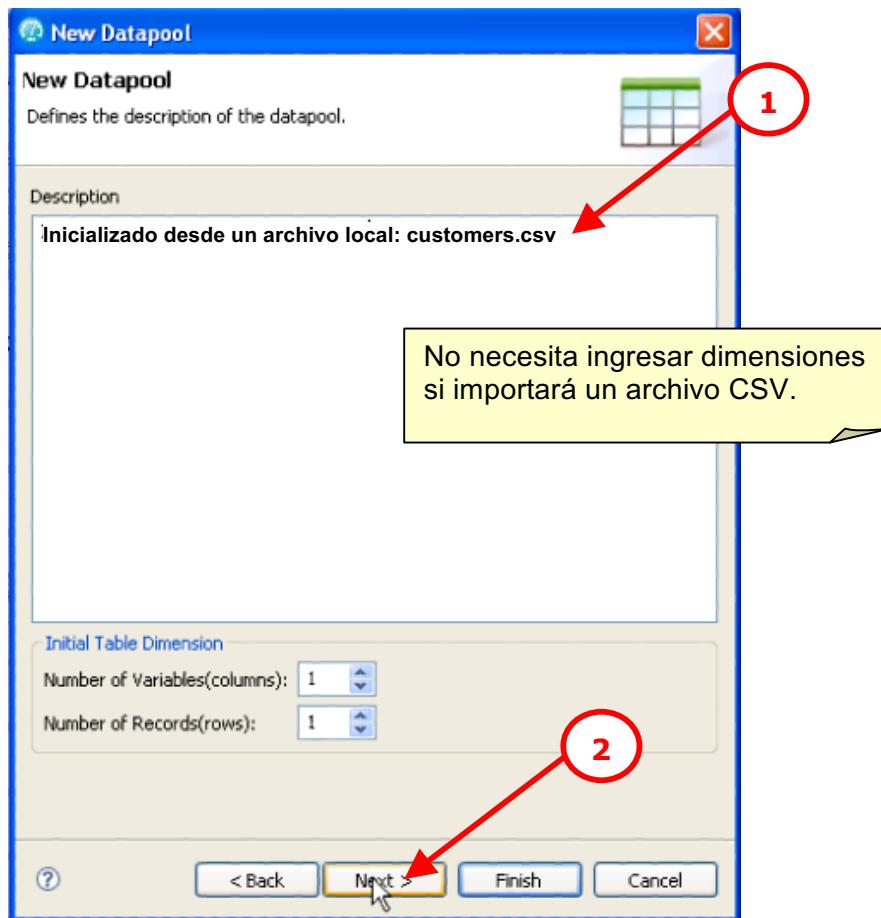
5. A continuación, active el *checkbox* para crear un nuevo pool de datos. Luego, pulse **Finish**.



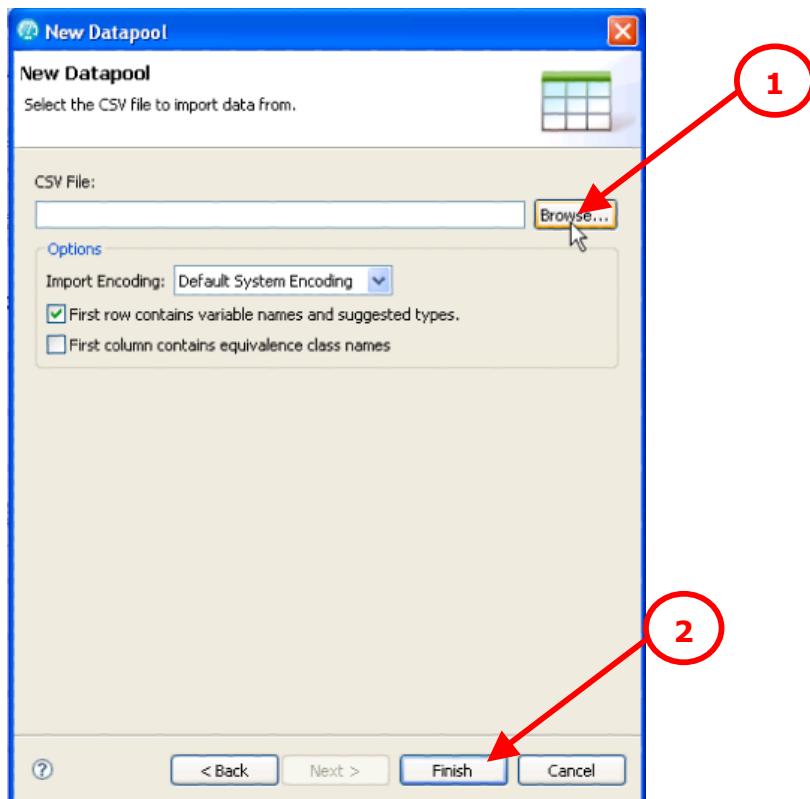
6. Desde la ventana **New Datapool** ingrese un nombre para el nuevo pool de datos. Luego, pulse **Next**.



7. A continuación, edite una descripción y luego, pulse **Next**.



8. Ubique el archivo CSV.



9. A continuación se mostrará el pool de datos importado.

The screenshot shows the Rational Performance Tester interface. The main window title is "Test - customers - Rational Performance Tester". The menu bar includes File, Edit, Navigate, Search, Project, Run, Window, Help. The toolbar has various icons for file operations. The left sidebar shows a project structure under "PlantsByWebsphere": "customers" and "order", with "order" expanded to show "order [August 3, 2007 12:00]". The central pane displays a "Datapool" table with two columns: "userid::" and "passwd::". The data rows are:

userid::	passwd::
jennifer@new.com	jennifer
1 john@new.com	john
2 melinda@new.com	melinda
3 nick@new.com	nick
4 pat@new.com	pat
5 phil@new.com	phil
6 richard@new.com	richard
7 wayne@new.com	wayne

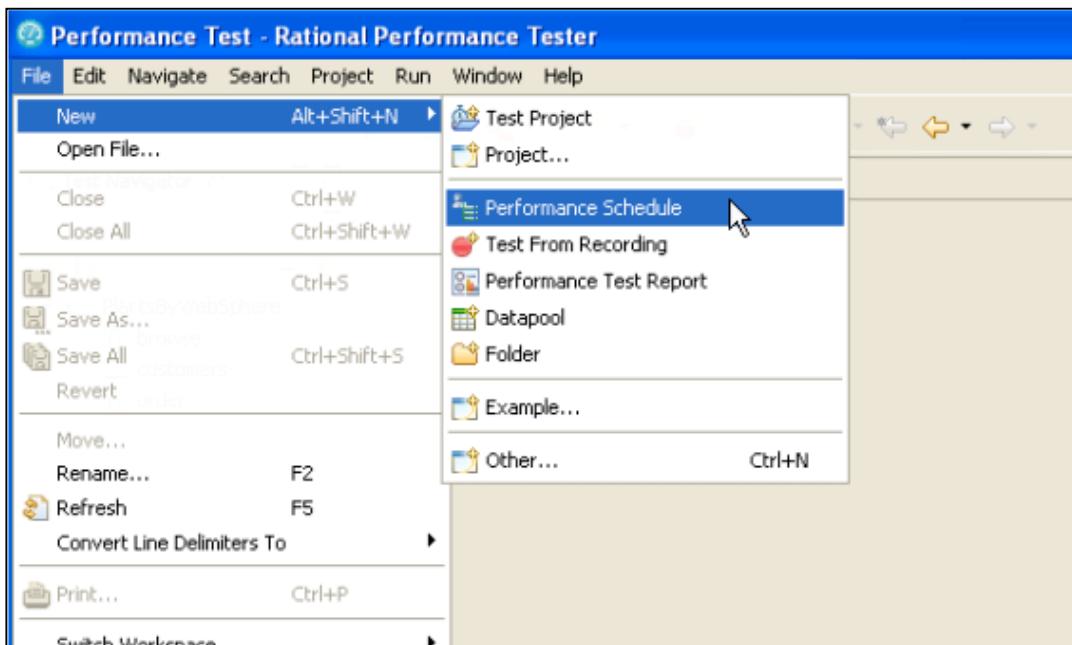
A yellow callout box points to the table with the text: "El **datapool** importado contiene los valores para 8 usuarios: usuario y password."

The bottom pane shows tabs for "Overview" and "EquivalenceClass1". The "Overview" tab is selected, showing a table with columns: Description, Resource, Path, and Location. The table is currently empty ("0 items").

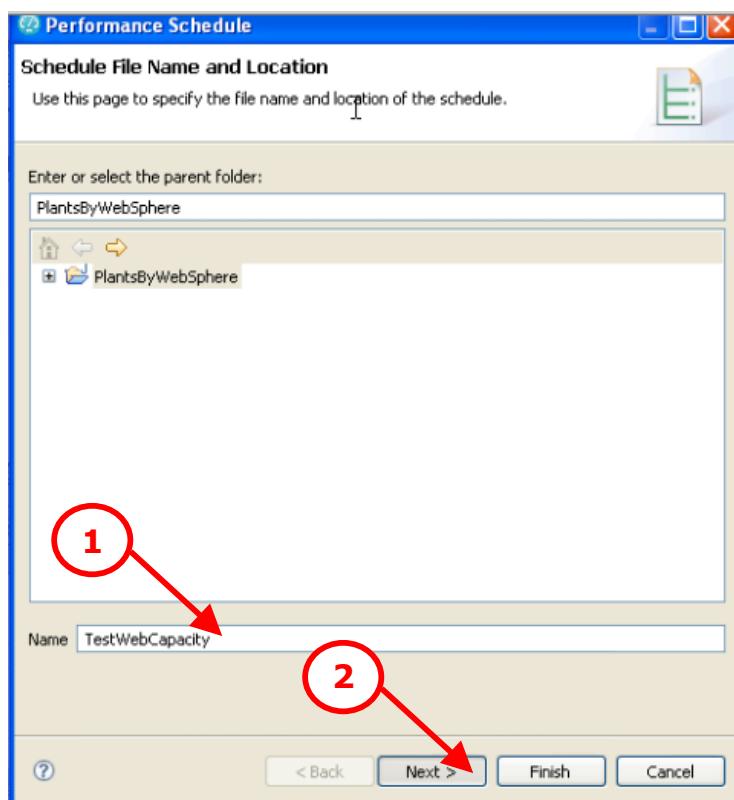
4. Programación de Pruebas

La programación de una prueba representa una carga de trabajo a fin de realizar las pruebas de rendimiento automatizadas. Los pasos se describen a continuación.

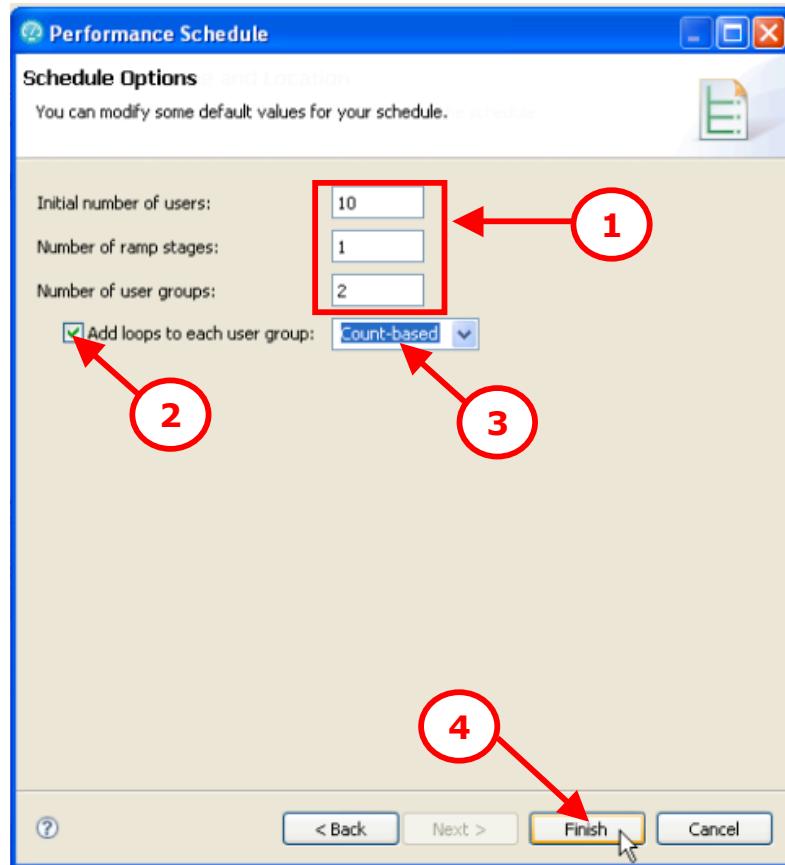
1. Seleccione **File > New > Performance Schedule** para programar una prueba.



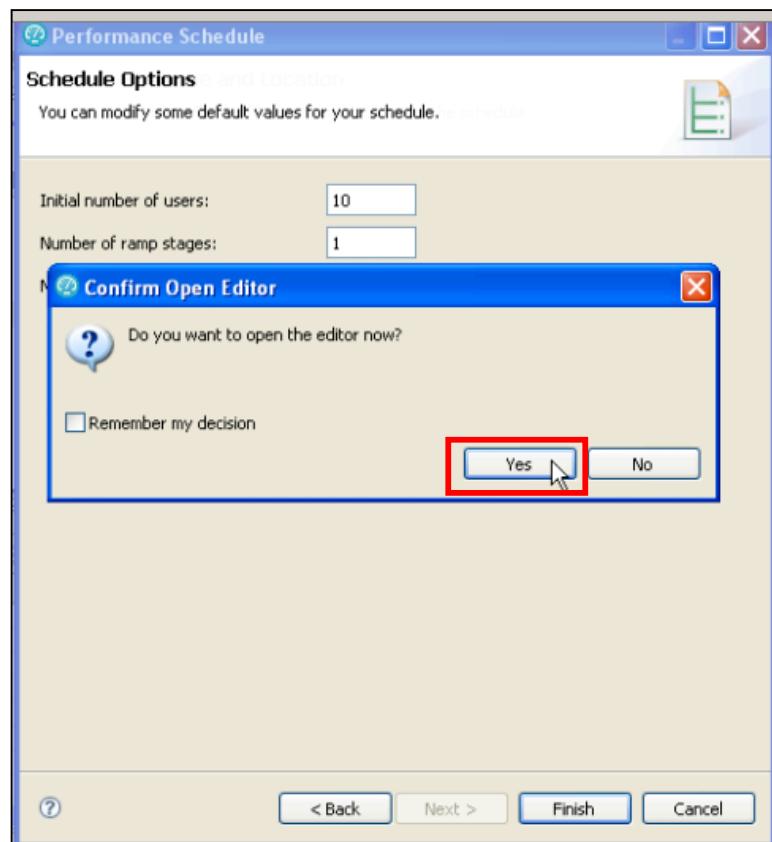
2. Ingrese el nombre de la programación y luego seleccione **Next**.



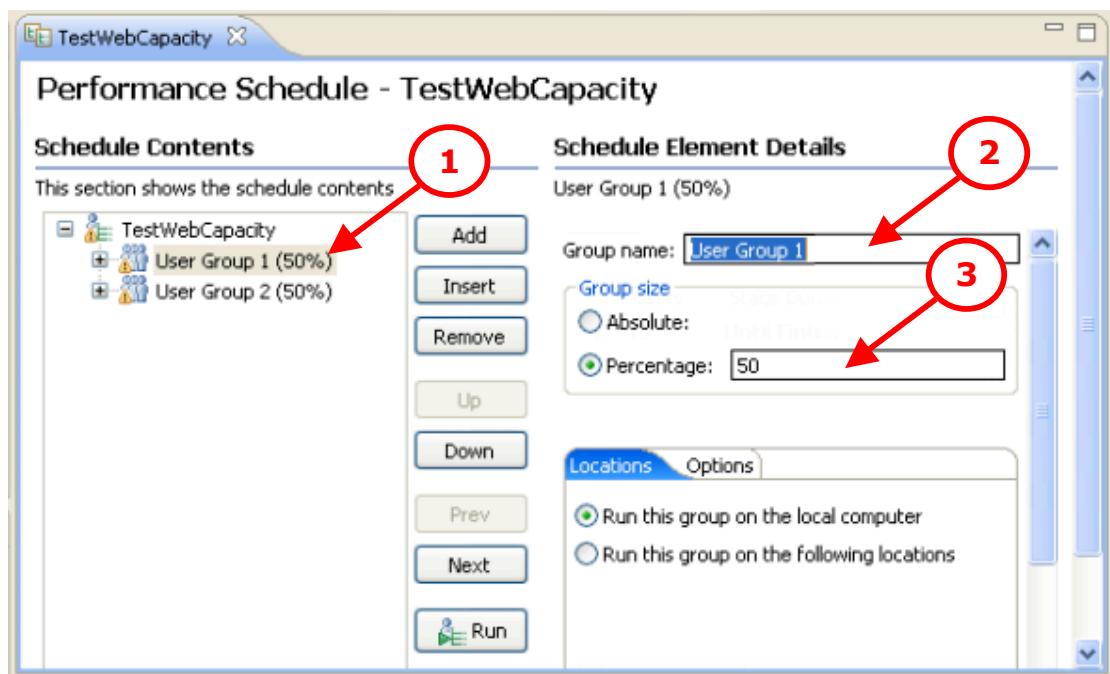
3. Ingrese los valores iniciales de la programación, seleccione el tipo de bucle para cada grupo de usuario y luego **Finish**.



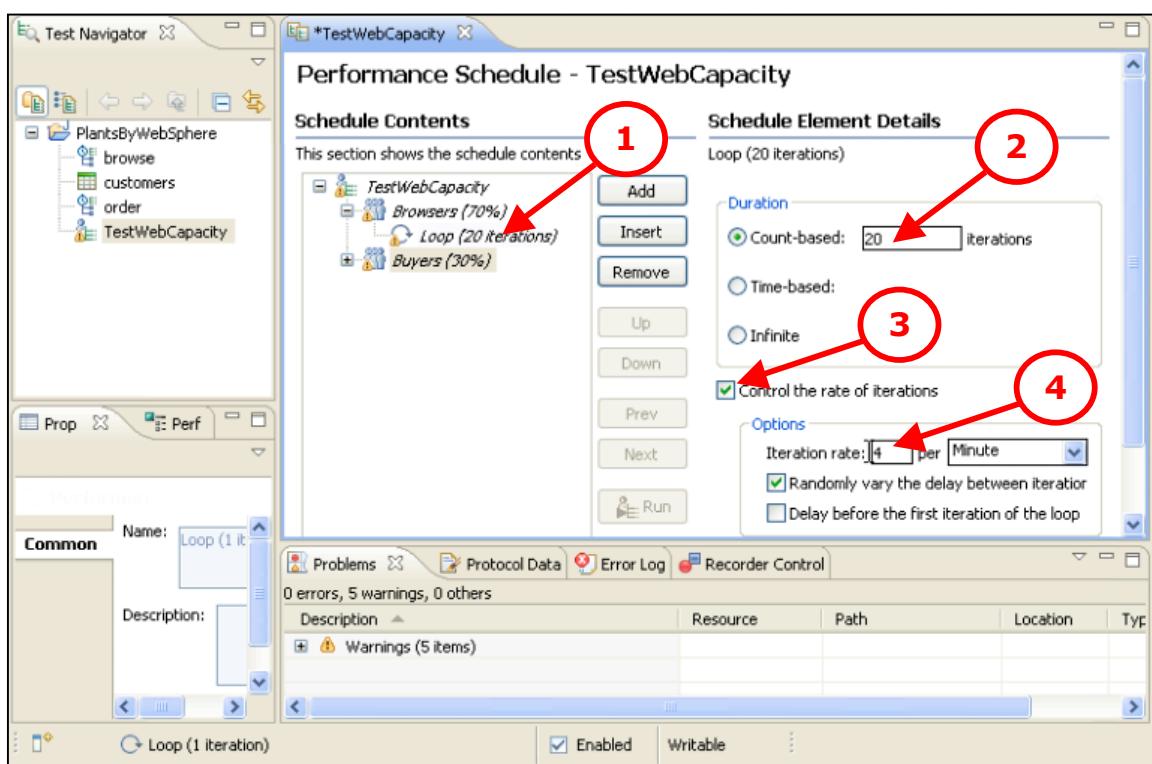
4. Ahora, confirme la visualización del editor de programación.



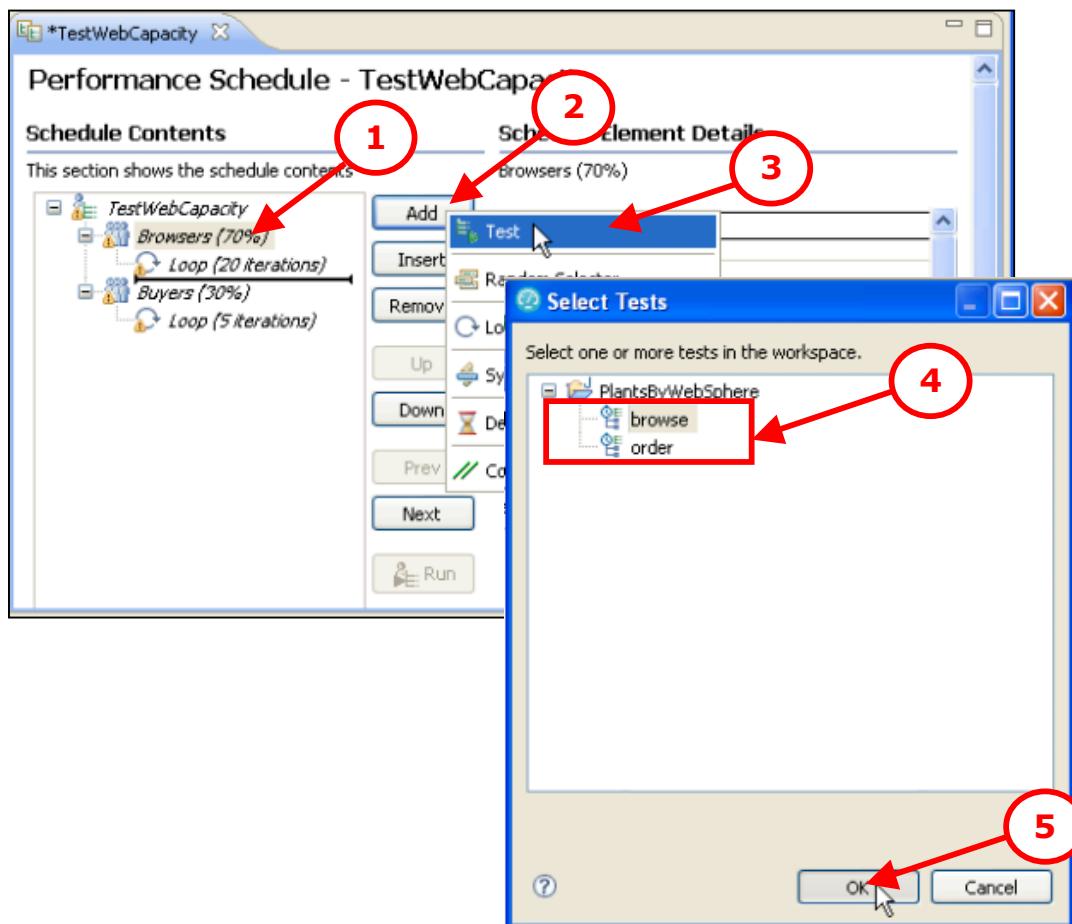
5. A continuación, modifique la información de cada grupo de usuario: nombre y porcentaje, según los usuarios que tendrá la aplicación a probar. Para ello, seleccione el grupo y realice los cambios desde la sección de detalles (panel izquierdo).



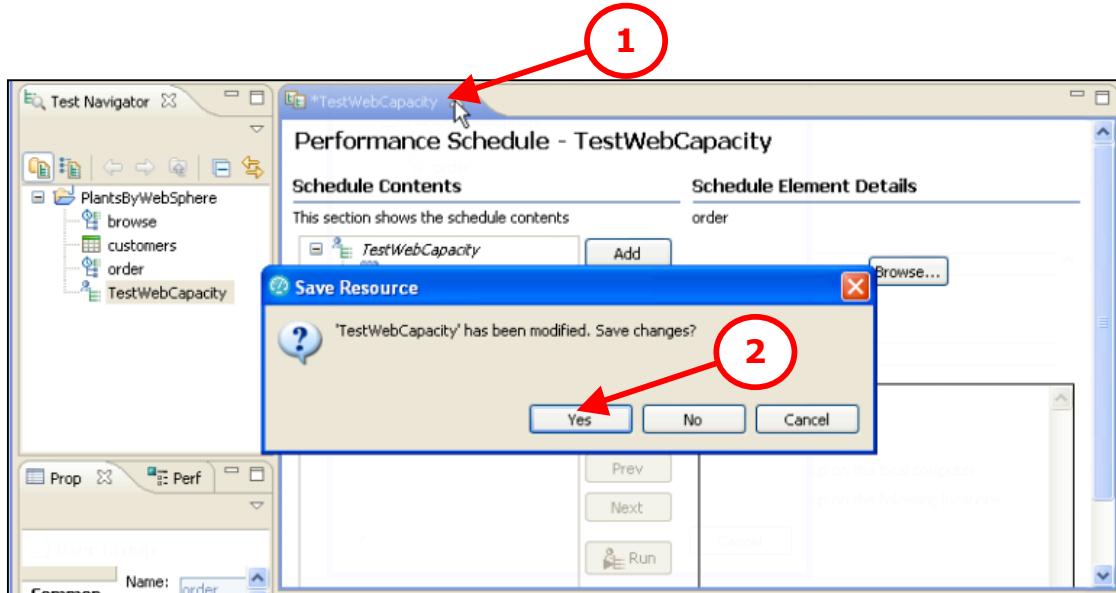
6. Ahora, agregue las características del bucle de cada grupo de usuario.



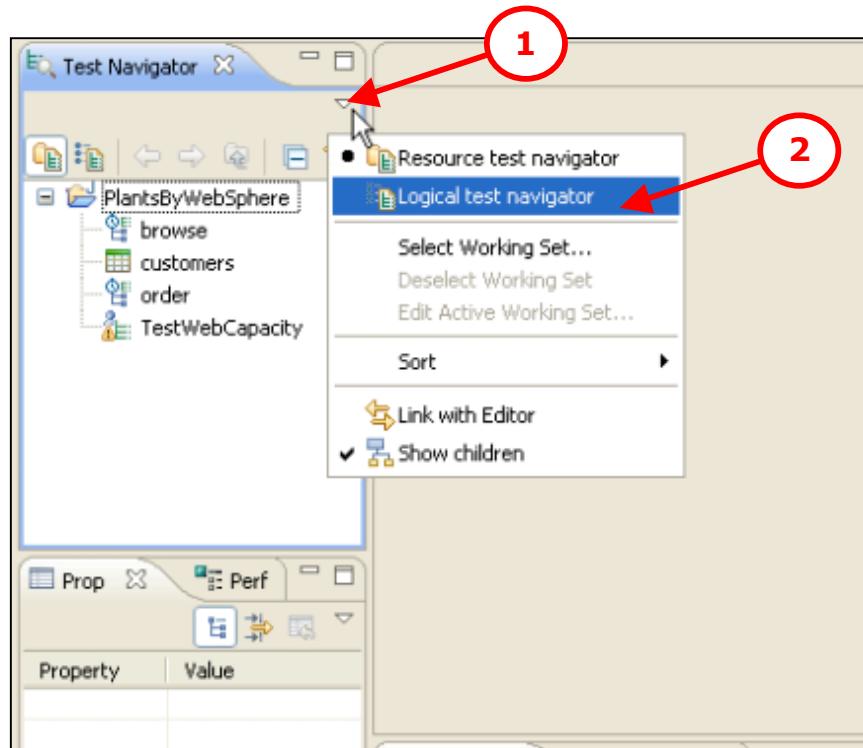
7. A continuación, agregue las pruebas que se requieran a cada grupo de usuario. Así:



8. Luego, cierre el editor de programación de pruebas y confirme para guardar los cambios.



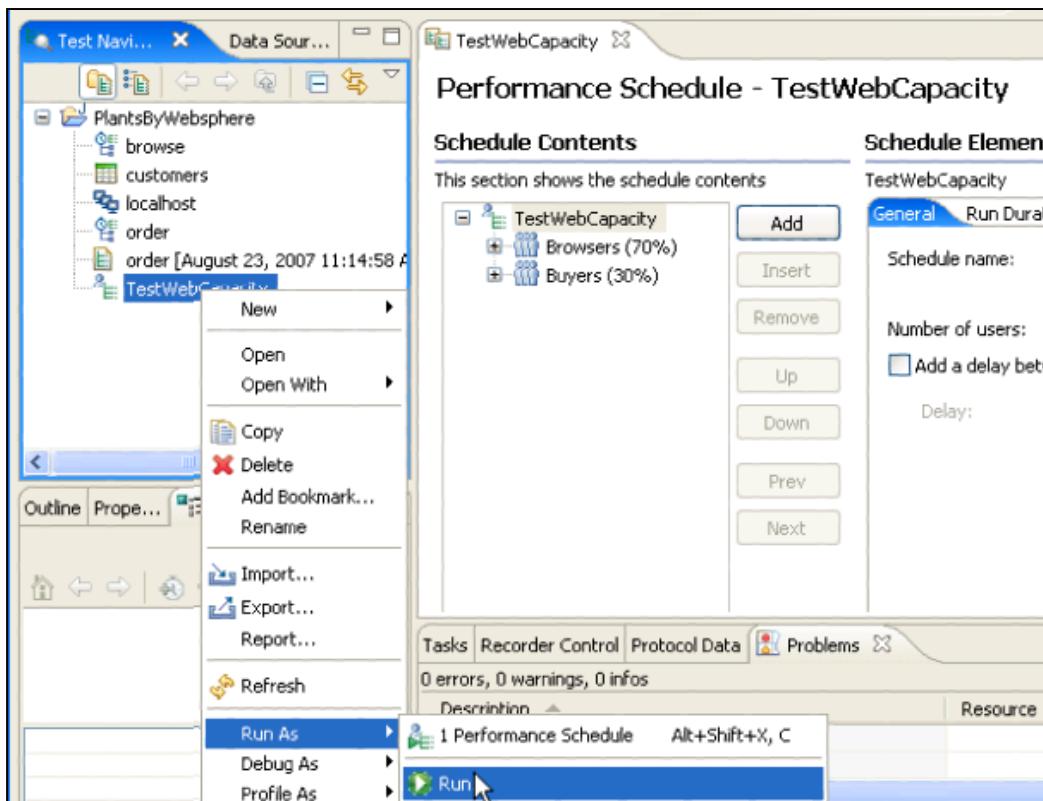
9. Por último, para agrupar los activos de prueba por tipo, seleccione el ícono del triángulo y seleccione **Logical test navigator**.



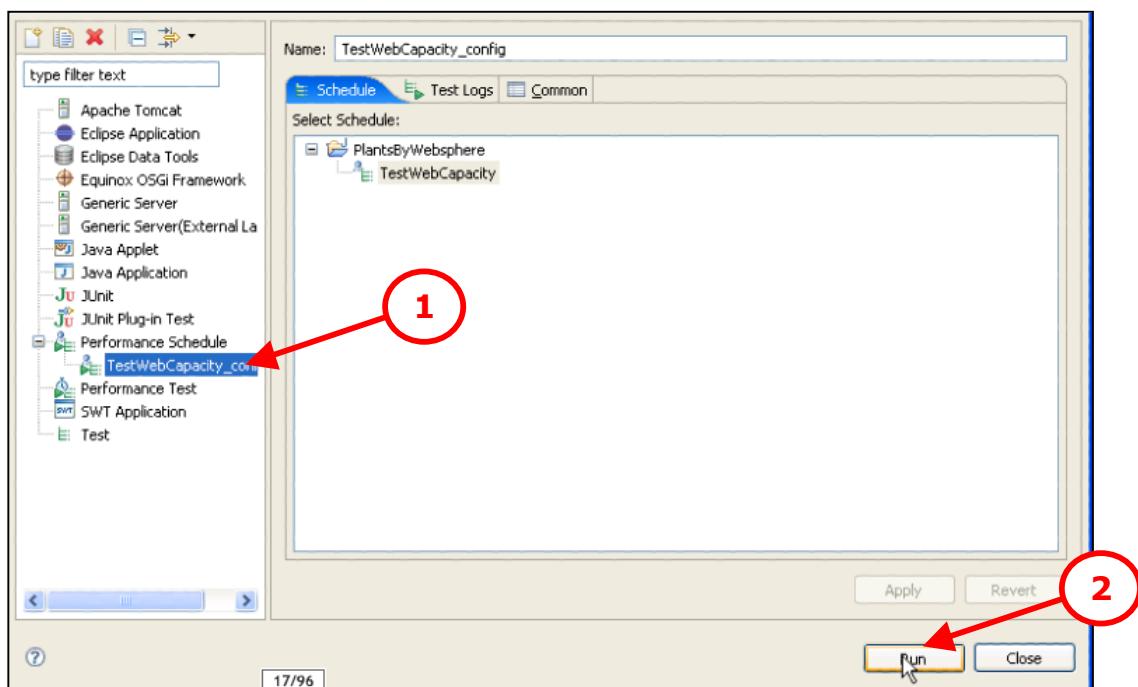
5. Ejecución de una Prueba

Para ejecutar la prueba con una programación previamente configurada realice los siguientes pasos:

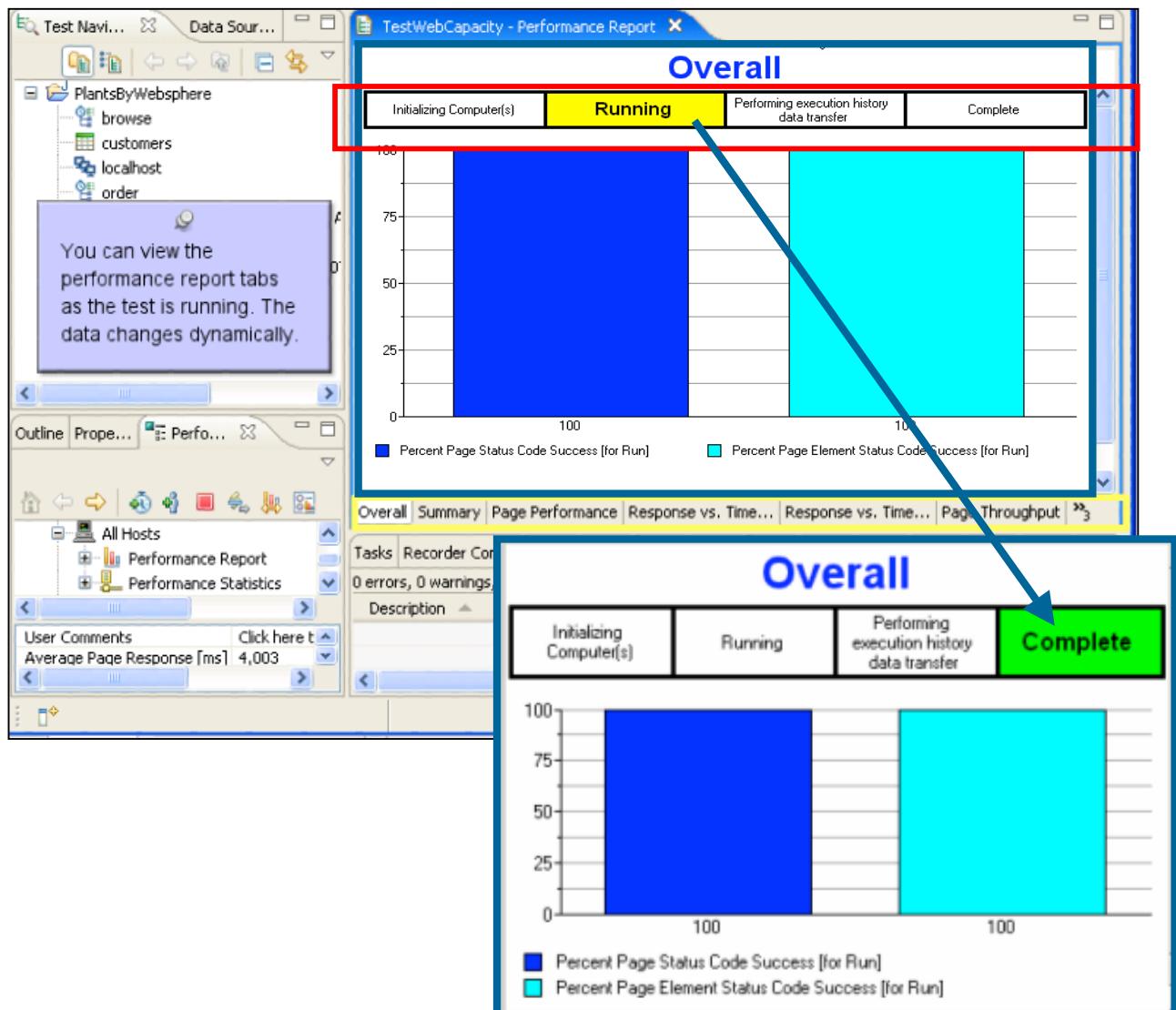
1. Seleccione ejecutar con la opción **Run**.



2. Seleccione una programación configurada localmente, de lo contrario se genera una programación por defecto.



3. Los informes de la programación, que se explicaron anteriormente, se visualizan e irán actualizándose en tiempo real **hasta que el indicador de progreso se complete**.



Resumen

- ॥ Los objetivos de las pruebas de rendimiento son:
 - Determinar los tiempos de respuesta del sistema
 - Determinar el número máximo de usuarios de un sistema (componentes, transacción, o configuración)
 - Descubrir las configuraciones óptimas o mínima del sistema
- ॥ Rational Performance Tester es una solución de verificación de cargas y rendimiento para equipos que se ocupen de la capacidad de ampliación de sus aplicaciones basadas en web. Gracias a la combinación de funciones de análisis detallados y fáciles de utilizar, Rational Performance Tester simplifica la creación de pruebas, la generación de cargas y la recopilación de datos para garantizar que las aplicaciones se amplíen hasta miles de usuarios concurrentes.
- ॥ Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.
 - ☞ <http://publib.boulder.ibm.com/infocenter/rpthelp/v8r1m0/index.jsp>
En esta página encontrará información del entorno IBM RPT: guía de instalación, guía de aprendizaje, entre otros temas.
 - ☞ http://agile.csc.ncsu.edu/SEMaterials/tutorials/rpt/index.html#section8_0
En esta página, hallará un ejercicio sobre prueba de rendimiento con RPT.
 - ☞ <http://www.ibm.com/developerworks/ssa/rational/library/09/buildingscriptsinrationalperformancetester/index.html>
En esta página, hallará un ejemplo de generación de scripts sólidos para pruebas de confiabilidad en RPT.